Transactions 2

CS348 Spring 2024

Instructor: Sujaya Maiyya

Sections: **002 & 003 only**

Outline For Today

1. Motivation For Transactions
2. ACID Properties
3. Different Levels of Isolation Beyond Serializability

Last lecture:
User's Perspective

Serializability:

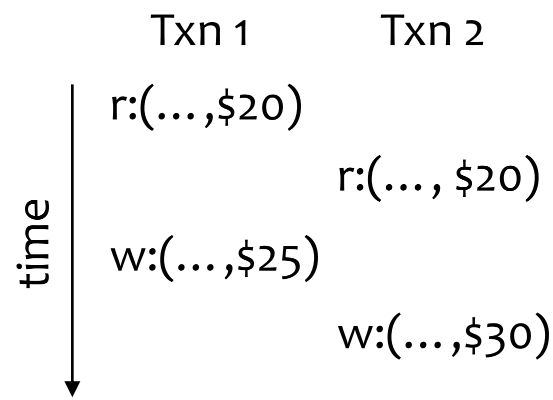
- Execution Histories
- Conflict Equivalence
- Checking For Conflict Equivalence

Concurrency control

Today's lecture:
System's Perspective

Goals of Execution History Model & Conflict Equivalences

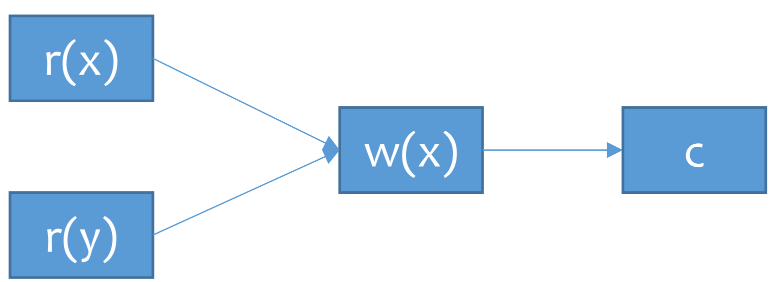
- Concurrency is achieved by interleaving operations across txns.



- Q: Does any interleaving correspond to a serializable execution?
- Execution history model and conflict equivalences is a formal method to answer this question.

Representing Single Transactions

- Database is a set of *data items* (often will denote as x, y, z…)
- Txn T_i is a *total order* of read/write & commit/abort operations on items
  - $r_i(x)$ indicates T_i reads item x
  - $w_i(x)$ indicates T_i writes item x
  - 'c' indicates commit ('a' indicates aborts)
    - Suppose: T_i does the following in this *chronological order*:
      - Read(x), Read(y), x ← x + y, Write(x), commit
      - $T_i=\{r_i(x) < r_i(y) < w_i(x) < c_i\}$ or simply as:
      - $T_i=\{r_i(x), r_i(y), w_i(x), c_i\}$ or $r_i(x), r_i(y), w_i(x), c_i$
- DAG representation



Execution histories (or schedules)

- An execution history over a set of transactions $T_1 \ldots T_n$ is an interleaving of the operations of $T_1 \ldots T_n$ in which the operation ordering imposed by each transaction is preserved.

- Two important assumptions:
  - Transactions interact with each other only via reads and writes of objects
  - A database is *a set* of objects

- Example: $T_1 = \{w_1[x], w_1[y], \mathrm{c}_1\}$, $T_2 = \{ r_2[x], r_2[y], c_2 \}$
  - $H_a = w_1[x]r_2[x]w_1[y]r_2[y]\mathrm{c}_1\mathrm{c}_2$
  - $H_b = w_1[x]w_1[y]\mathrm{c}_1 r_2[x]r_2[y]c_2$
  - $H_c = w_1[x]r_2[x]r_2[y]w_1[y]\mathrm{c}_1\mathrm{c}_2$ [next slide expands this example]
  - $H_d = r_2[x]r_2[y]c_2\ w_1[x]w_1[y]c_1$

Examples for valid execution history

- $T_1 = \{w_1[x], w_1[y], c_1\}, T_2 = \{ r_2[x], r_2[y], c_2 \}$

T_1	T_2
w1(x)	
	r2(x)
w1(y)	
	r2(y)
c1	
	c2

H_a

T_1	T_2
w1(x)	
w1(y)	
c1	
	r2(x)
	r2(y)
	c2

H_b

T_1	T_2
w1(x)	
	r2(x)
	r2(y)
w1(y)	
c1	
	c2

H_c

T_1	T_2
	r2(x)
	r2(y)
	c2
w1(x)	
w1(y)	
c1	

H_d

Examples for invalid execution history

$T_1 = \{w_1[x], w_1[y], c_1\}, T_2 = \{ r_2[x], r_2[y], c_2 \}$

T_1	T_2
w1(y)	
	r2(x)
w1(x)	
	r2(y)
c1	
	c2

T_1	T_2
w1(x)	
	r2(y)
w1(y)	
	r2(x)
c1	
	c2

Incorrect orders

Serial execution histories

- $T_1 = \{w_1[x], w_1[y], c_1\}, T_2 = \{\, r_2[x], r_2[y], c_2 \,\}$
- Serial histories: no interleaving operations from different txns

T_1	T_2
w1(x)	
	r2(x)
w1(y)	
	r2(y)
c1	
	c2

H_a

T_1	T_2
w1(x)	
w1(y)	
c1	
	r2(x)
	r2(y)
	c2

H_b ✓

T_1	T_2
w1(x)	
	r2(x)
	r2(y)
w1(y)	
c1	
	c2

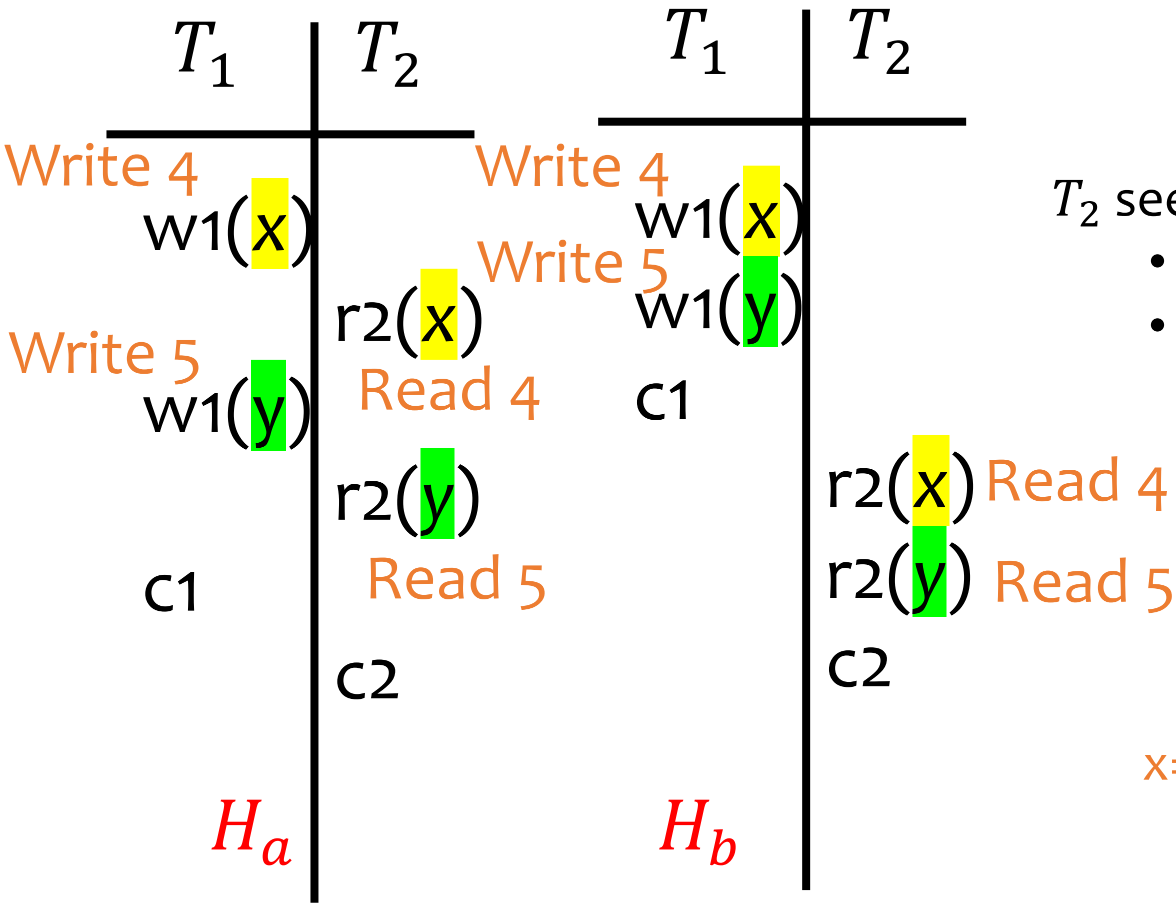
H_c

T_1	T_2
	r2(x)
	r2(y)
	c2
w1(x)	
w1(y)	
c1	

H_d ✓

Equivalent histories

- H_a is “equivalent” to H_b (a serial execution)



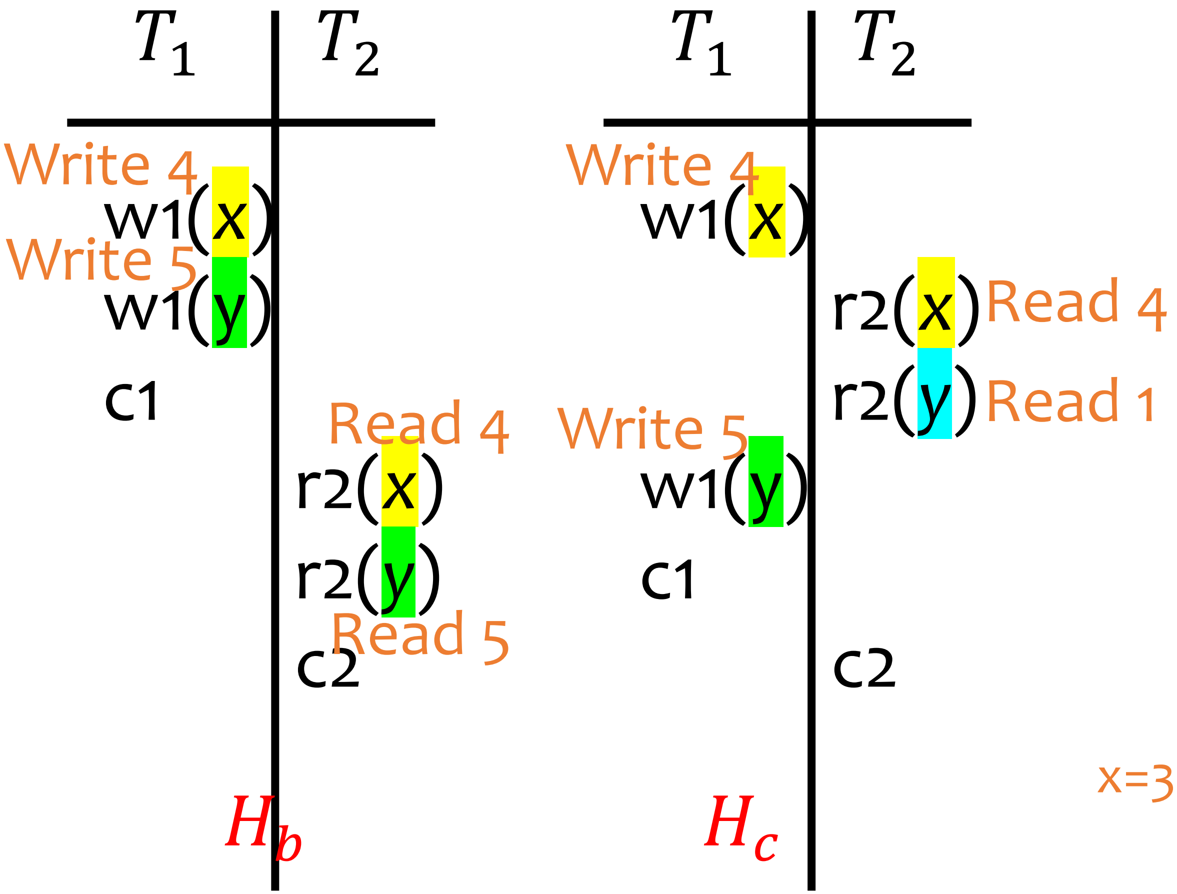
T_2 sees all the updates made by T_1

- T_2 reads x written by T_1
- T_2 reads y written by T_1

x=3, y=1 (before T1 and T2)

Equivalent histories

- H_c is not "equivalent" to H_b (a serial execution)



T_2 reads different y in H_b as in H_c

x=3, y=1 (before T1 and T2)

Outline For Today

Serializability:

1. Execution Histories
2. Conflict Equivalence
3. Checking For Serializability

Concurrency control:

1. 2 phase locking

Check equivalence

- Two operations conflict if:
  1. they belong to **different transactions**,
  2. they operate on the **same object**, and
  3. at least one of the operations is a **write**

2 types of conflicts: (1) Read-Write (or write-read) and (2) Write-Write

- Two histories are conflict equivalent if
  1. they are over the same set of transactions, and
  2. the ordering of each pair of conflicting operations is the same in each history

Example

- Consider
  - $H_a = w_1[x]r_2[x]w_1[y]r_2[y]c_1c_2$
  - $H_b = w_1[x]w_1[y]r_2[x]r_2[y]c_1c_2$

Step 1: check if they are over the same set of transactions

- $T_1 = \{w_1[x], w_1[y]\}, T_2 = \{ r_2[x], r_2[y] \}$

Step 2: check if all the conflicting pairs have the same order

Conflicting pairs	H_a	H_b
$w_1[x], r_2[x]$	$<$	$<$
$w_1[y], r_2[y]$	$<$	$<$

Motivation & Intuition For Conflict Equivalence

- If two histories H_a and H_b are conflict equivalent then, we can make H_a exactly the same as H_b by iteratively swapping two consecutive non-conflicting operations in H_a and/or H_b.
  - $H_a = w_1[x]r_2[x]w_1[y]r_2[y]c_1c_2 \Rightarrow H'_a = w_1[x]w_1[y]r_2[x]r_2[y]c_1c_2$
  - $H_b = w_1[x]w_1[y]r_2[x]r_2[y]c_1c_2$
- Proof Sketch: Move all ops on item x to the beginning by swapping with non-conflicting ops in both H_a and H_b
- End with the order imposed by the conflicts on x
- If H_a & H_b are conflict eq. this prefix ops on x will be the same order
- Then repeat for y, z, etc. and we will arrive at the same histories
- Therefore: Every read by each txn has the same value in H_a & H_b
- Therefore: H_a & H_b lead to the same output database state.

More complicated example

Consider

- H_A: $r_1[x]r_3[x]w_4[y]r_2[u]w_4[z]r_1[y]r_3[u]r_2[z]w_2[z]r_3[z]r_1[z]w_3[y]$
- H_B: $r_1[x]w_4[y]r_3[x]r_2[u]r_1[y]r_3[u]r_2[z]w_2[z]w_4[z]r_1[z]r_3[z]w_3[y]$

Step 1: check if they are over the same set of transactions

Step 2: check if all the conflicting pairs have the same order

More complicated example

Consider

- H_A: $r_1[x]r_3[x]w_4[y]r_2[u]w_4[z]r_1[y]r_3[u]r_2[z]w_2[z]r_3[z]r_1[z]w_3[y]$
- H_B: $r_1[x]w_4[y]r_3[x]r_2[u]r_1[y]r_3[u]r_2[z]w_2[z]w_4[z]r_1[z]r_3[z]w_3[y]$

Step 1: check if they are over the same set of transactions

T$_1$: $\{r_1[x]\ r_1[y]\ r_1[z]\}$, **T$_2$**: $\{r_2[u]\ r_2[z]w_2[z]\}$,
T$_3$: $\{r_3[x]\ r_3[u]\ r_3[z]w_3[y]\}$, **T$_4$**: $\{w_4[y]\ w_4[z]\}$

Step 2: check if all the conflicting pairs have the same order

Identify all the conflicting pairs

- H_A: $r_1[x]r_3[x]w_4[y]r_2[u]w_4[z]r_1[y]r_3[u]r_2[z]w_2[z]r_3[z]r_1[z]w_3[y]$

- Conflicting pairs:
  - Related to x or u: no conflicting pairs, as all are reads
  - Related to y: w4[y], r1[y], w3[y]
    - $w_4[y] < r_1[y]$
    - $w_4[y] < \mathrm{w}_3[y]$
    - $r_1[y] < \mathrm{w}_3[y]$
  - Related to z: w4[z], r2[z], w2[z], r3[z], r1[z]
    - $w_4[z] < r_2[z]$
    - $w_4[z] < w_2[z]$
    - $w_4[z] < r_3[z]$
    - $w_4[z] < r_1[z]$
    - $r_2[z]$, $w_2[z]$ are not, as they are from the same transactions
    - $\mathrm{w}_2[z] < r_3[z]$
    - $\mathrm{w}_2[z] < r_1[z]$

More complicated example

Consider

- H_A: $r_1[x]r_3[x]w_4[y]r_2[u]w_4[z]r_1[y]r_3[u]r_2[z]w_2[z]r_3[z]r_1[z]w_3[y]$
- H_B: $r_1[x]w_4[y]r_3[x]r_2[u]r_1[y]r_3[u]r_2[z]w_2[z]w_4[z]r_1[z]r_3[z]w_3[y]$

Step 1: check if they are over the same set of transactions

T_1: $\{r_1[x]\ r_1[y]\ r_1[z]\ \}$, **$T_2$**: $\{r_2[u]\ r_2[z]w_2[z]\}$,
T_3: $\{r_3[x]\ r_3[u]\ r_3[z]w_3[y]\}$, **$T_4$**: $\{w_4[y]\ w_4[z]\}$

Step 2: check if all the conflicting pairs have the same order

Conflicting pairs	H_A	H_B
$w_4[y], r_1[y]$	$<$	$<$
$w_4[y], w_3[y]$	$<$	$<$
…	$<$	$<$
$w_4[z], w_2[z]$	$<$	$>$
…	$<$	$<$

Outline For Today

Serialization:

1. Execution Histories
2. Conflict Equivalence
3. Checking For Serializability

Concurrency control:

1. 2 phase locking

Serializable

- A history H is said to be (conflict) **serializable** if there exists some serial history H' that is (conflict) equivalent to H.

T_1	T_2
w1(x)	
	r2(x)
w1(y)	
	r2(y)
c1	
	c2

T_1	T_2
w1(x)	
w1(y)	
c1	
	r2(x)
	r2(y)
	c2

$H_a = H_b$

T_1	T_2
w1(x)	
	r2(x)
	r2(y)
w1(y)	
c1	
	c2

H_c ?

Serializable

- Does H_c have an equivalent serial execution?
  - $H_c = w_1[x]r_2[x]r_2[y]w_1[y]c_1c_2$
- Only 2 serial execution to check:
  - H_b: T_1 followed by T_2: $w_1[x]w_1[y]c_1r_2[x]r_2[y]c_2$
    - $r_2[y]$ reads different value as in H_c
  - H_d: T_2 followed by T_1: $r_2[x]r_2[y]c_2w_1[x]w_1[y]c_1$
    - $r_2[x]$ reads different value as in H_c

Conflicting pairs	H_b	H_c	H_d
$w_1[x], r_2[x]$	<	<	>
$w_1[y], r_2[y]$	<	>	>

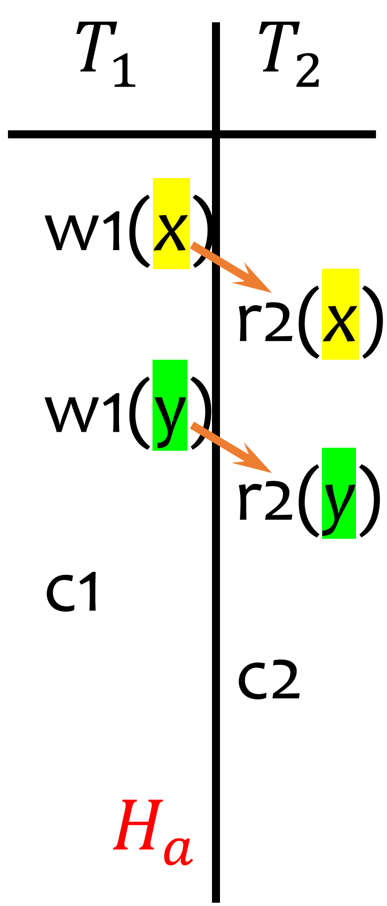
- Do we need to check all the serial executions?

How to test for serializability?

- Serialization graph $SG_H(V, E)$ for history H:
  - $V = \{T | T \text{ is a committed transaction in } H\}$
  - $E = \{T_i \rightarrow T_j$ if $o_i \in T_i$ and $o_j \in T_j$ **conflict** and $o_i < o_j\}$
- A history is serializable iff its serialization graph is acyclic.

Example

- Example: $H_a = w_1[x] r_2[x] w_1[y] r_2[y]\ c_1 c_2$

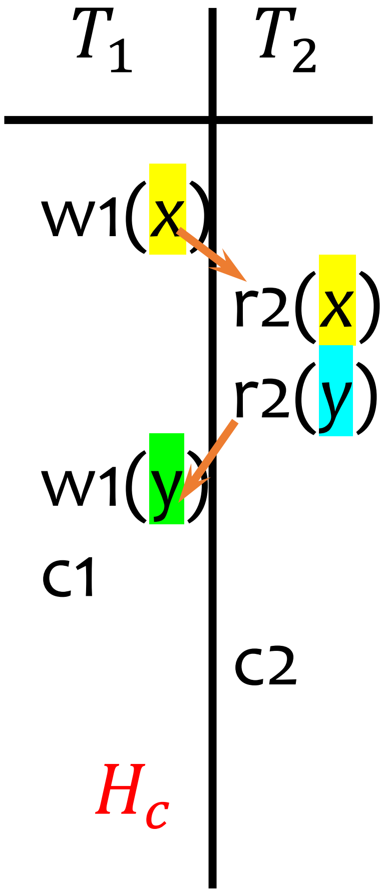


$w_1[x]$ and $r_2[x]$ conflict, and $w_1[x] < r_2[x]$
$w_1[y]$ and $r_2[y]$ conflict, and $w_1[y] < r_2[y]$

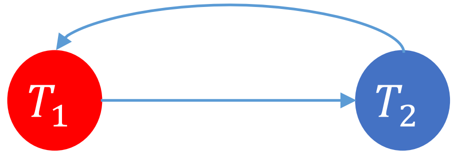
Serialization graph: no cycles → serializable

Example

- Example: $H_c = w_1[x] r_2[x] r_2[y] w_1[y] c_1 c_2$



$w_1[x]$ and $r_2[x]$ conflict, and $w_1[x] < r_2[x]$;
$w_1[y]$ and $r_2[y]$ conflict, and $r_2[y] < w_1[y]$

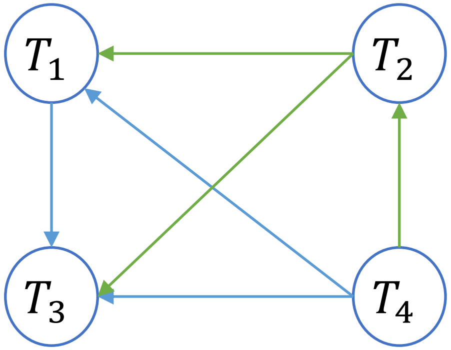


Not serializable

More complicated example

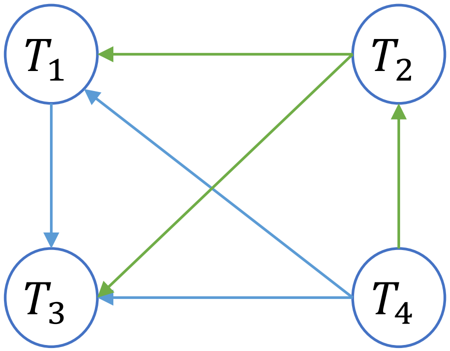
- $r_1[x]r_3[x]w_4[y]r_2[u]w_4[z]r_1[y]r_3[u]r_2[z]w_2[z]r_3[z]r_1[z]w_3[y]$

- Conflicting pairs:
  - Related to x or u: no conflicting pairs, as all are reads
  - Related to y: w4[y], r1[y], w3[y]
    - $w_4[y] < r_1[y]$ T4 → T1
    - $w_4[y] < w_3[y]$ T4 → T3
    - $r_1[y] < w_3[y]$ T1 → T3
  - Related to z: w4[z], r2[z], w2[z], r3[z], r1[z]
    - $w_4[z] < r_2[z]$ T4 → T2
    - $w_4[z] < w_2[z]$ T4 → T2
    - $w_4[z] < r_3[z]$ T4 → T3
    - $w_4[z] < r_1[z]$ T4 → T1
    - $r_2[z]$, $w_2[z]$ are not, as they are from the same transactions
    - $w_2[z] < r_3[z]$ T2 → T3
    - $w_2[z] < r_1[z]$ T2 → T1



More complicated example

- $r_1[x]r_3[x]w_4[y]r_2[u]w_4[z]r_1[y]r_3[u]r_2[z]w_2[z]r_3[z]r_1[z]w_3[y]$



- No cycles in this serialization graph
  - Topological sort: T4 -> T2 -> T1->T3

- The history above is (conflict) equivalent to

$$w_4[y]w_4[z]\ r_2[u]r_2[z]w_2[z]\ r_1[x]r_1[y]r_1[z]\ r_3[x]r_3[u]r_3[z]w_3[y]$$

  - Note: we ignore the commits at the end for simplicity

Outline For Today

Serializability:

1. Execution Histories
2. Conflict Equivalence
3. Checking For Serializability

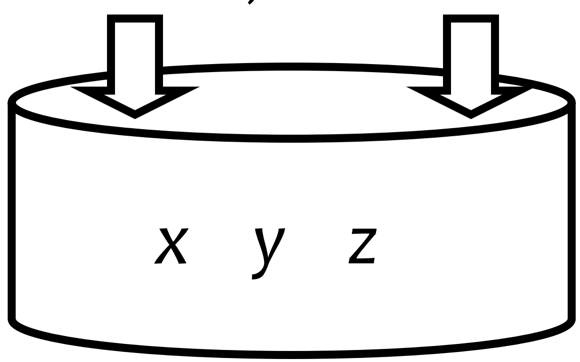
Concurrency control:

1. 2 phase locking

Concurrency control

- Goal: ensure the "I" (isolation) in ACID

T_1:	T_2:
r1(x);	r2(x);
w1(x);	w2(x);
r1(y);	r2(z);
w1(y);	w2(z);
commit;	commit;



Good versus bad execution histories

Serial

Good!

T_1	T_2
r1(x)	
w1(x)	
r1(y)	
w1(y)	
	r2(x)
	w2(x)
	r2(z)
	w2(z)

H_a

Bad!

	T_1	T_2	
Read 400	r1(x)		
		r2(x)	Read 400
Write 400 – 100	w1(x)		
		w2(x)	Write 400 – 50
	r1(y)		
		r2(z)	
	w1(y)		
		w2(z)	

H_b

Good! Why?

Hint: construct serialization graph

T_1	T_2
r1(x)	
w1(x)	
	r2(x)
	w2(x)
r1(y)	
	r2(C)
w1(y)	
	w2(C)

H_c

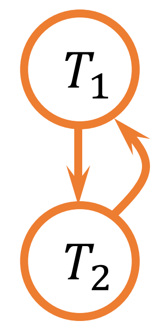
Good versus bad execution histories

Not serializable

Bad!

T_1	T_2
r1(x)	
	r2(x)
w1(x)	
	w2(x)
r1(y)	
	r2(z)
w1(y)	
	w2(z)

H_b



How to avoid this?

Note: These are 'valid' histories but are 'bad': cannot be serialized

Concurrency control

Possible classification

- Pessimistic – assume that conflicts will happen and take preventive action
  - Two-phase locking (2PL)

- Optimistic – assume that conflicts are rare and run transactions and fix if there is a problem
  - Timestamp ordering

- We will only review 2PL

Locking

- Rules
  - If a transaction wants to read an object, it must first request a shared lock (S mode) on that object
  - If a transaction wants to modify an object, it must first request an exclusive lock (X mode) on that object
  - Allow one exclusive lock, or multiple shared locks

Mode of the lock requested

Mode of lock(s) currently held by other transactions

	S	X
S	Yes	No
X	No	No

Grant the lock?

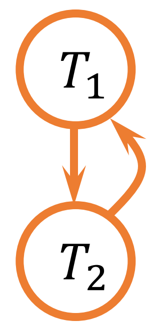
Compatibility matrix

Basic locking is not enough

Possible schedule under locking

But still not conflict-serializable!

T_1	T_2
lock-X(x)	
r1(x)	
w1(x)	
unlock(x)	
	lock-X(x)
	r2(x)
	w2(x)
	unlock(x)
	lock-X(y)
	r2(y)
	w2(y)
	unlock(y)
lock-X(y)	
r1(y)	
w1(y)	
unlock(y)	

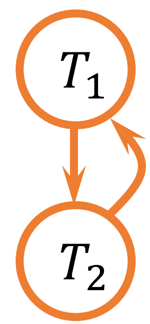


Basic locking is not enough

Add 1 to both *x* and *y* (preserve *x*=*y*)	T_1	T_2	Multiply both *x* and *y* by 2 (preserves *x*=*y*)
	lock-X(*x*)		
Read 100	r1(*x*)		
Write 100+1	w1(*x*)		
	unlock(*x*)		
		lock-X(*x*)	
		r2(*x*)	Read 101
		w2(*x*)	Write 101*2
		unlock(*x*)	
		lock-X(*y*)	
		r2(*y*)	Read 100
		w2(*y*)	Write 100*2
		unlock(*y*)	
	lock-X(*y*)		
Read 200	r1(*y*)		
Write 200+1	w1(*y*)		
	unlock(*y*)		

Possible schedule under locking

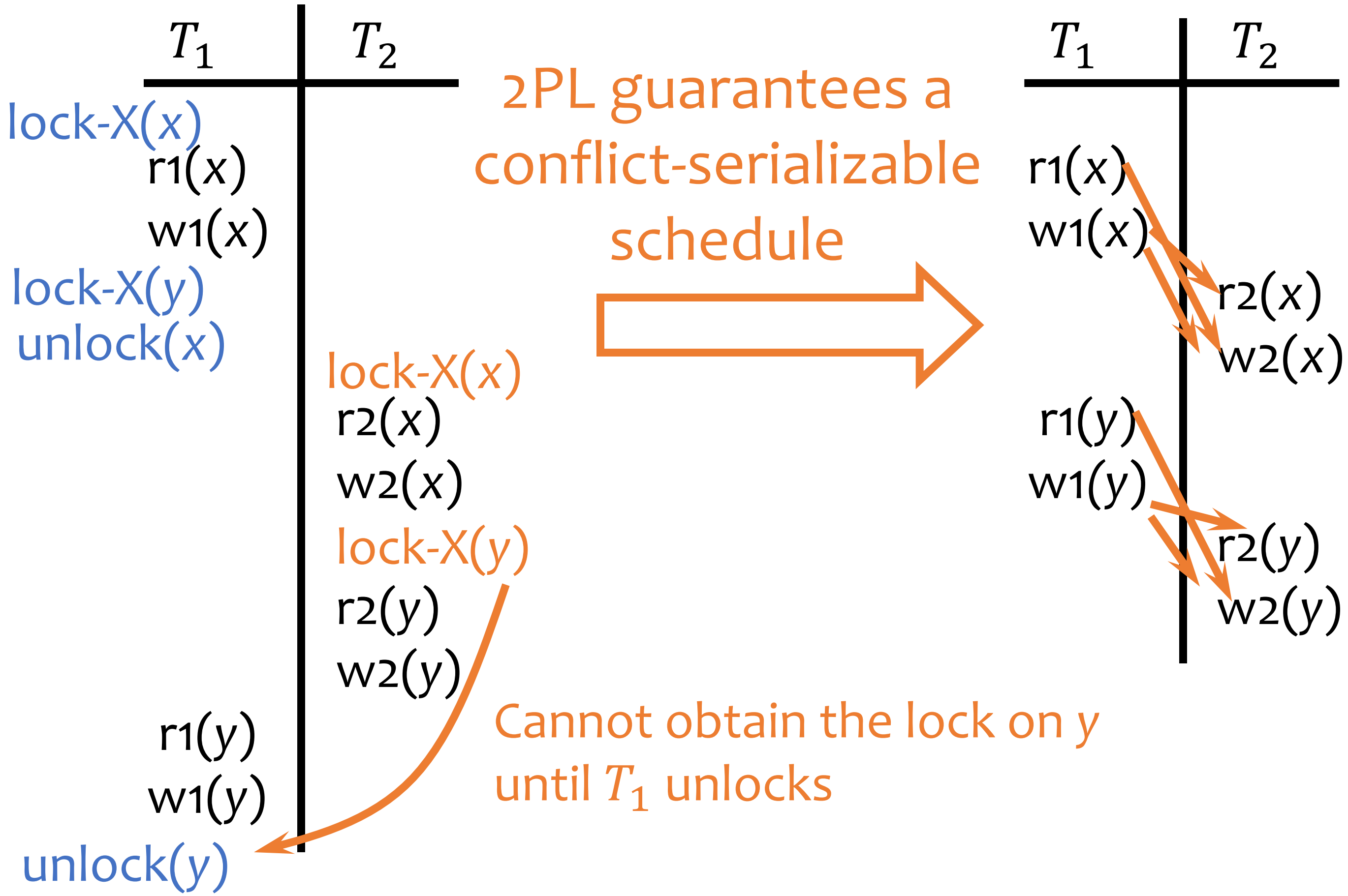
But still not conflict-serializable!



$x \neq y$!

Two-phase locking (2PL)

- All lock requests precede all unlock requests
  - Phase 1: obtain locks, phase 2: release locks



Remaining problems of 2PL

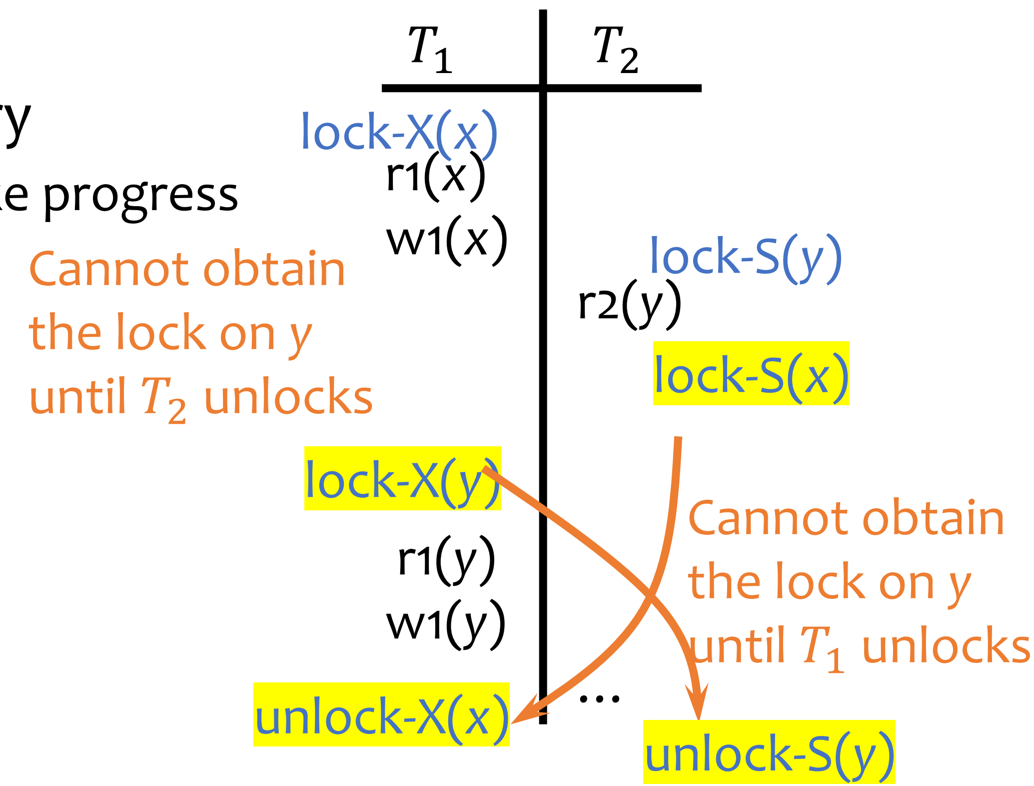
T_1	T_2
r1(x)	
w1(x)	
	r2(x)
	w2(x)
r1(y)	
w1(y)	
	r2(y)
	w2(y)
Abort!	

- T_2 has read uncommitted data written by T_1
- If T_1 aborts, then T_2 must abort as well
- Cascading aborts possible if other transactions have read data written by T_2

- Even worse, what if T_2 commits before T_1?
  - Schedule is not recoverable if the system crashes right after T_2 commits

Deadlocks

- A transaction is deadlocked if it is blocked and will remain blocked until there is an intervention.
- Locking-based concurrency control algorithms may cause deadlocks requiring abort of one of the transactions
- Consider the partial history
  - Neither T_1 nor T_2 can make progress



Strict 2PL

- Only release X-locks at commit/abort time
  - A writer will block all other readers until the writer commits or aborts
- Used in many commercial DBMS
  - Avoids cascading aborts
  - But deadlocks are still possible!
- Conservative 2PL: acquire all locks at the beginning of a txn
  - Avoids deadlocks but often not practical

Summary

Serializability:

1. Execution Histories
2. Conflict Equivalence
3. Checking For Serializability

Concurrency control:

1. 2 phase locking