

# SQL: Transactions

Introduction to Database Management

CS348 Fall 2022

# Announcements (Thur., Nov 17)

- **Project**

- **Milestone 2** due **Nov 17** (Thu)
- **Final demo** in the week of **Nov 25<sup>th</sup> – Dec 1<sup>st</sup> (Week 13)**
  - Email your TA the choice of your demo (online/video) **by Nov 24**
  - Lose points if failing to do so
  - No lecture in that week
- **Final report** is due Dec 1<sup>st</sup> (Thu)

- **Assignment 3**

- Cover Lectures 11-15
- Due Nov 24 (Thu)

- **Final exam**

- **Final Review Session** (online), on Tue Dec 6<sup>th</sup>
- Exam on Dec 13 (7:30pm – 10pm)
- Cover Lectures 1- 15

# Why we need transactions

- A database is a **shared** resource accessed by many users and processes **concurrently**.
  - Both queries and modifications
- Not managing this concurrent access to a shared resource will cause problems (not unlike in operating systems)
  - Problems due to **concurrency**
  - Problems due to **failures**

# Problems caused by concurrency

- Accounts(Anum, CId, BranchId, Balance)
- Application 1: You are depositing money to your bank account.

**update** Accounts  
**set** Balance = Balance + 100  
**where** Anum = 9999

- Application 2: The branch is calculating the balance of the accounts.

**select** Sum(Balance)  
**from** Accounts

- Problem – Inconsistent reads
  - If the applications run concurrently, the total balance returned to application 2 may be inaccurate

# Another concurrency problem

- Application 1: You are depositing money to your bank account at an ATM.

**update** Accounts  
**set** Balance = Balance + 100  
**where** Anum = 9999

- Application 2: Your partner is withdrawing money from the same account at another ATM.

**update** Accounts  
**set** Balance = Balance – 50  
**where** Anum = 9999

- Problem – Lost Updates
  - If the applications run concurrently, one of the updates may be “lost”, and the database may be inconsistent.

# Yet another concurrency problem

- Application 1:

```
update Employee  
set Salary = Salary + 1000  
where WorkDept = 'D11'
```

- Application 2:

```
select * from Employee  
where WorkDept = 'D11'
```

```
select * from Employee  
where Lastname like 'A%'
```



- Problem – Non-Repeatable Reads

- If there are employees in D11 with surnames that begin with “A”, Application 2’s queries may see them with different salaries.

# High-level lesson

- We need to worry about interaction between two applications when
  - one **reads** from the database while the other writes to (modifies) the database; or
  - both **write** to (modify) the database.
- We do **not** worry about interaction between two applications when both only read from the database.

# Problems caused by failures

- Update all account balances at a bank branch.  
**update** Accounts  
**set** Balance = Balance \* 1.05  
**where** BranchId = 12345
- Problem: If the system crashes **while** processing this update, some, but not all, tuples with BranchId = 12345 (i.e., some account balances) may have been updated.
- Problem: If the system crashes **after** this update is processed but before all of the changes are made permanent (updates may be happening in the buffer), the changes may not survive.



# Another failure-related problem

- transfer money between accounts:

**update** Accounts

**set** Balance = Balance – 100

**where** Anum = 8888

**update** Accounts

**set** Balance = Balance + 100

**where** Anum = 9999

- Problem: If the system fails between these updates, money may be withdrawn but not redeposited.

# High-level lesson

- We need to worry about **partial** results of applications on the database when a crash occurs.
- We need to make sure that when applications are **completed** their changes to the database survive crashes.

# Transactions

- A **transaction** is a sequence of database operations with the following properties (**ACID**):
  - **Atomic**: Operations of a transaction are **executed all-or-nothing**, and are never left “half-done”
  - **Consistency**: Assume all database **constraints** are satisfied at the start of a transaction, they should **remain satisfied at the end of the transaction**
  - **Isolation**: Transactions must behave as if they were executed in **complete isolation from each other**
  - **Durability**: If the DBMS **crashes** after a transaction commits, **all effects** of the transaction must **remain** in the database when DBMS comes back up

# SQL transactions

- A transaction is automatically started when a user executes an SQL statement
- Subsequent statements in the same session are executed as part of this transaction
  - Statements **see changes** made by earlier ones **in the same transaction**
  - Statements in **other concurrently running transactions** do **not**
- **COMMIT** command commits the transaction
  - Its effects are made final and visible to subsequent transactions
- **ROLLBACK** command aborts the transaction
  - Its effects are undone

# Fine prints

- Schema operations (e.g., CREATE TABLE) implicitly commit the current transaction
  - Because it is often difficult to undo a schema operation
- Many DBMS support an **AUTOCOMMIT** feature, which automatically commits every single statement
  - You can turn it on/off through the API
  - For PostgreSQL:
    - psql command-line processor turns it on by default
    - You can turn it off at the psql prompt by typing:  
`\set AUTOCOMMIT 'off'`

# Atomicity

- Partial effects of a transaction must be undone when
  - User explicitly aborts the transaction using ROLLBACK
  - The DBMS crashes before a transaction commits
- Partial effects of a modification statement must be undone when any constraint is violated
  - Some systems roll back only this statement and let the transaction continue; others roll back the whole transaction
- How is atomicity achieved?
  - Logging (to support undo) – next lecture [optional]

# Durability

- DBMS accesses data on stable storage by bringing data into memory
- Effects of committed transactions must survive DBMS crashes
- How is durability achieved?
  - Forcing all changes to disk at the end of every transaction?
    - Too expensive
  - Logging (to support redo) – next lecture [optional]

# Consistency

- Guaranteed by constraints and triggers declared in the database and/or transactions themselves
- Whenever inconsistency arises,
  - abort the statement or transaction, or
  - fix the inconsistency within the transaction



# Isolation (focus of this lecture)

- Transactions must appear to be executed in a **serial schedule** (with no interleaving operations)
- For performance, DBMS executes transactions using a **serializable schedule**
  - In this schedule, operations from different transactions can interleave and execute concurrently
  - But the schedule is guaranteed to produce the same effects as a serial schedule
- How is isolation achieved?
  - Locking, multi-version concurrency control, etc. (next lecture) [optional]

# Outline

- Transactions
  - Properties: ACID
- Isolation
  - Different isolation levels
    - Based on allowed anomalies: dirty reads, non-repeatable reads, phantoms
    - Serializability (focus)
  - How to set isolation level for transactions?

# SQL isolation levels

- Strongest isolation level: **SERIALIZABLE**
  - Complete isolation
- Weaker isolation levels: **REPEATABLE READ**, **READ COMMITTED**, **READ UNCOMMITTED**
  - Increase performance by eliminating overhead and allowing higher degrees of concurrency
  - Trade-off: sometimes you get the “wrong” answer

# READ UNCOMMITTED

- Can read “dirty” data
  - A data item is dirty if it is written by an uncommitted transaction
- Problem: What if the transaction that wrote the dirty data eventually aborts?
- Example: wrong average
  - -- T1:  
UPDATE User  
SET pop = 0.99  
WHERE uid = 142;  
  
ROLLBACK;
  - -- T2:  
  
SELECT AVG(pop)  
FROM User;  
  
COMMIT;

# READ COMMITTED

- No dirty reads, but **non-repeatable reads** possible
  - Reading the same data item twice can produce different results
- Example: different averages

- -- T1:

```
UPDATE User
SET pop = 0.99
WHERE uid = 142;
COMMIT;
```

- T2:

```
SELECT AVG(pop)
FROM User;
```

```
SELECT AVG(pop)
FROM User;
COMMIT;
```

# REPEATABLE READ

- Reads are repeatable, but may see **phantoms**
- Example: different average (still!)

- -- T1:

```
INSERT INTO User  
VALUES(789, 'Nelson',  
      10, 0.1);  
COMMIT;
```

- -- T2:

```
SELECT AVG(pop)  
FROM User;
```

```
SELECT AVG(pop)  
FROM User;  
COMMIT;
```

# Serializable

- All the three anomalies should be avoided:
  - Dirty reads
  - Unrepeatable reads
  - Phantoms
- Serial executions of T1 and T2 definitely prevent the three anomalies:
  - T1 followed by T2 or T2 followed by T1
- Can we run T1 and T2 concurrently and achieve the same serial effect?

# Outline

- Transactions
  - Properties: ACID
- Isolation
  - Different isolation levels
    - Based on allowed anomalies: dirty reads, non-repeatable reads, phantoms
    - Serializability (focus)
      - Execution history
      - Conflict equivalence to a serial execution history
      - How to check if a execution history is serializable
  - Which isolation level to choose for SQL transactions?



# Example for a single transaction

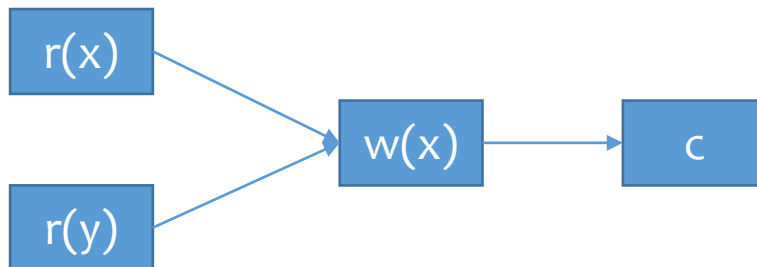
- Consider a transaction  $T$ :

$$T = \{Read(x), Read(y), x \leftarrow x + y, Write(x), commit\}$$

A set of operations:  $\{r[x], r[y], w[x], c\}$

A set of partial orders between operations:  $\{(r[x] < w[x]), (r[y] < w[x]), (r[x] < c), (r[y] < c), (w[x] < c)\}$

- DAG representation



# Transaction definition – formal

- Let
  - $o_i(x)$  be some operation of transaction  $T$  operating on data item  $x$ , where  $o_i \in \{read, write\}$  and  $o_i$  is atomic;
  - $OS = \{\cup o_i\}$ ;
  - $N \in \{abort, commit\}$
- Transaction  $T$  is a partial order  $T = \{\Sigma, <\}$ , where
  - $\Sigma = OS \cup \{N\}$
  - For any two operations  $o_i, o_j \in OS$ , if  $o_i = \{r(x) \text{ or } w(x)\}$  and  $o_j = w(x)$  for any data item  $x$ , then either  $o_i < o_j$  or  $o_j < o_i$
  - $\forall o_i \in OS, o_i < N$

# Execution histories

- An **execution history** over a set of transactions  $T_1 \dots T_n$  is an interleaving of the operations of  $T_1 \dots T_n$  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 fixed set of independent objects*
- Example:  $T_1 = \{w_1[x], w_1[y], c_1\}$ ,  $T_2 = \{r_2[x], r_2[y], c_2\}$ 
  - $H_a = w_1[x]r_2[x]w_1[y]r_2[y]c_1c_2$
  - $H_b = w_1[x]w_1[y]c_1r_2[x]r_2[y]c_2$
  - $H_c = w_1[x]r_2[x]r_2[y]w_1[y]c_1c_2$
  - $H_d = r_2[x]r_2[y]c_2 w_1[x]w_1[y]c_1$  [next slide expands this example]



# Examples for valid execution history

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

$T_1$	$T_2$	$T_1$	$T_2$	$T_1$	$T_2$	$T_1$	$T_2$
w1(x)		w1(x)		w1(x)			r2(x)
	r2(x)	w1(y)			r2(x)		r2(y)
w1(y)		c1			r2(y)		c2
	r2(y)		r2(x)	w1(y)		w1(x)	
c1			r2(y)	c1		w1(y)	
	c2		c2		c2	c1	
$H_a$		$H_b$		$H_c$		$H_d$	

# Serial execution histories

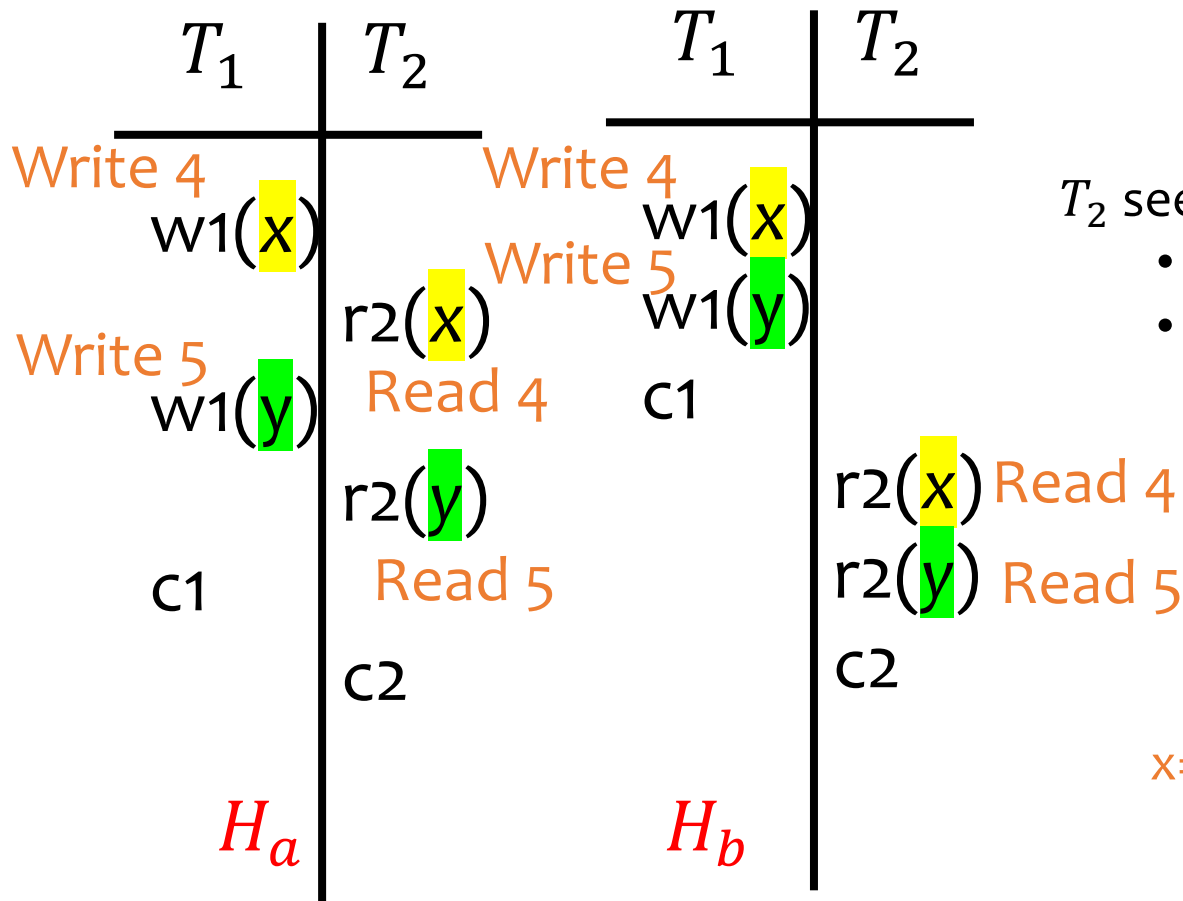
- $T_1 = \{w_1[x], w_1[y], c_1\}$ ,  $T_2 = \{r_2[x], r_2[y], c_2\}$

$T_1$	$T_2$	$T_1$	$T_2$	$T_1$	$T_2$	$T_1$	$T_2$
$w_1(x)$		$w_1(x)$		$w_1(x)$			$r_2(x)$
	$r_2(x)$	$w_1(y)$			$r_2(x)$		$r_2(y)$
$w_1(y)$		$c_1$			$r_2(y)$		$c_2$
	$r_2(y)$		$r_2(x)$	$w_1(y)$		$w_1(x)$	
$c_1$			$r_2(y)$	$c_1$		$w_1(y)$	
	$c_2$		$c_2$		$c_2$	$c_1$	
$H_a$		$H_b$		$H_c$		$H_d$	

Serial histories: no interleaving operations from different transactions

# Equivalent histories

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



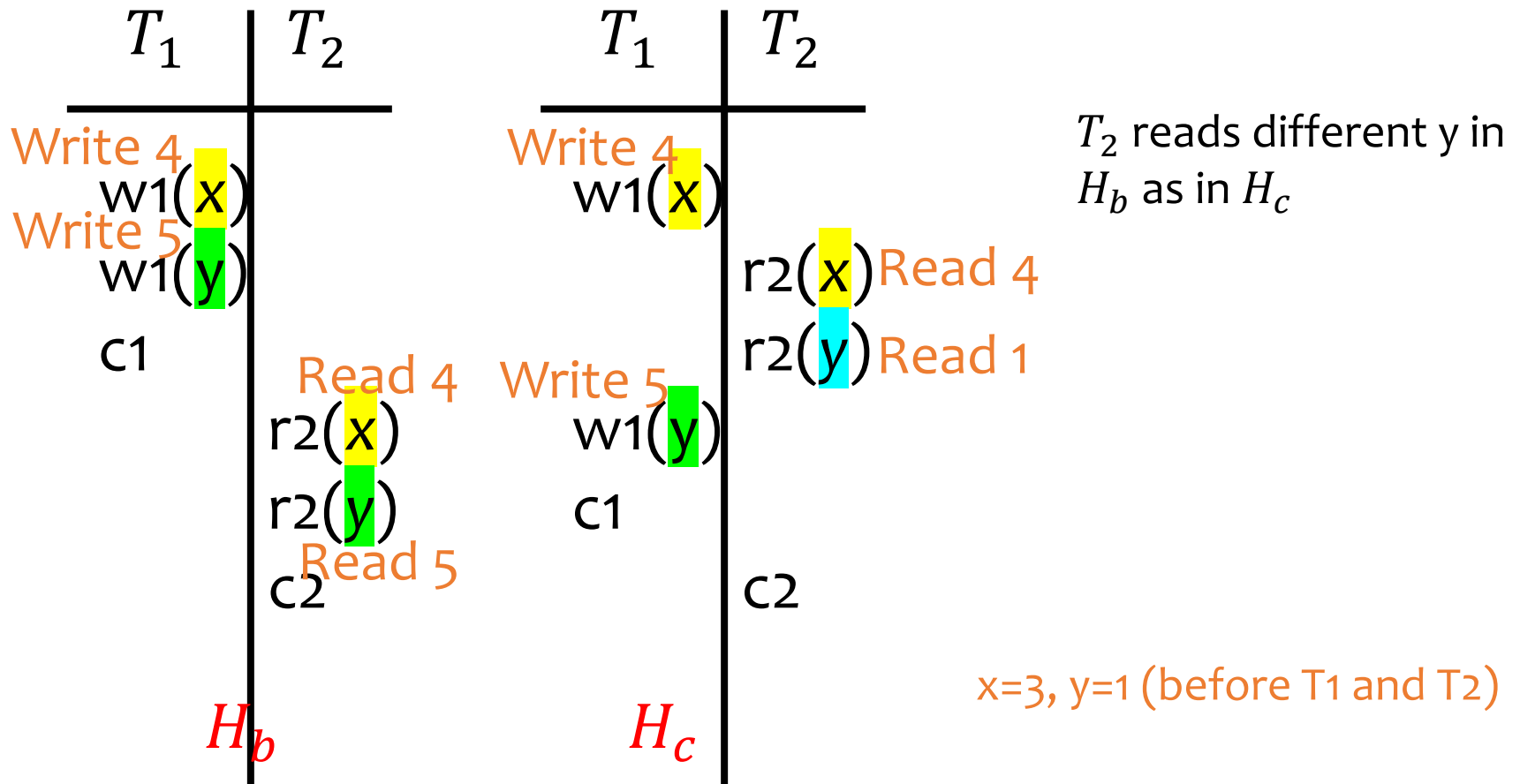
$T_2$  sees all the updates made by  $T_1$

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

$x=3, y=1$  (before  $T_1$  and  $T_2$ )

# Equivalent histories

- $H_c$  is not “equivalent” to  $H_b$  (a serial execution)



# 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 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	$H_a$	$H_b$
$w_1[x], r_2[x]$	<	<
$w_1[y], r_2[y]$	<	<

# 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

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

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: no conflicting pairs, as all are reads
  - Related to y:  $w_4[y]$ ,  $r_1[y]$ ,  $w_3[y]$ 
    - $w_4[y] < r_1[y]$
    - $w_4[y] < w_3[y]$
    - $r_1[y] < w_3[y]$
  - Related to z:  $w_4[z]$ ,  $r_2[z]$ ,  $w_2[z]$ ,  $r_3[z]$ ,  $r_1[z]$ 
    - $w_4[z] < r_2[z]$
    - $w_4[z] < w_2[z]$
    - $w_4[z] < r_3[z]$
    - $w_4[z] < r_1[z]$
    - $r_2[z]$ ,  $w_2[z]$  are not, as they are from the same transactions
    - $w_2[z] < r_3[z]$
    - $w_2[z] < r_1[z]$

# 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

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

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

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

# 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
$H_a = H_b$	

$T_1$	$T_2$
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?

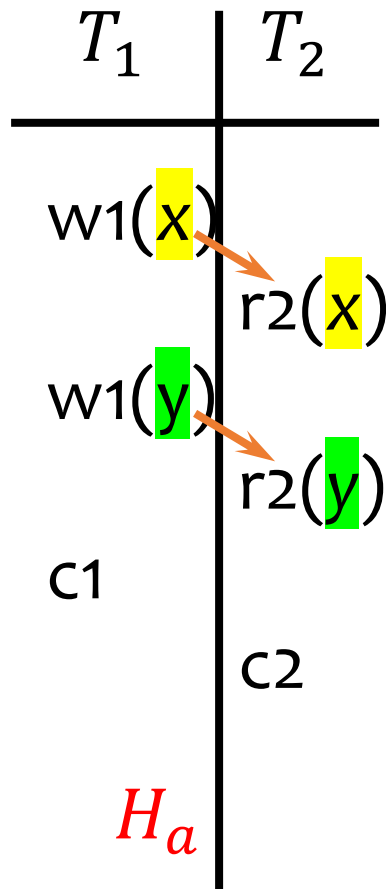
# How to test for serializability?

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



# Example

- Example:  $H_a = w_1[x]r_2[x]w_1[y]r_2[y] c_1c_2$



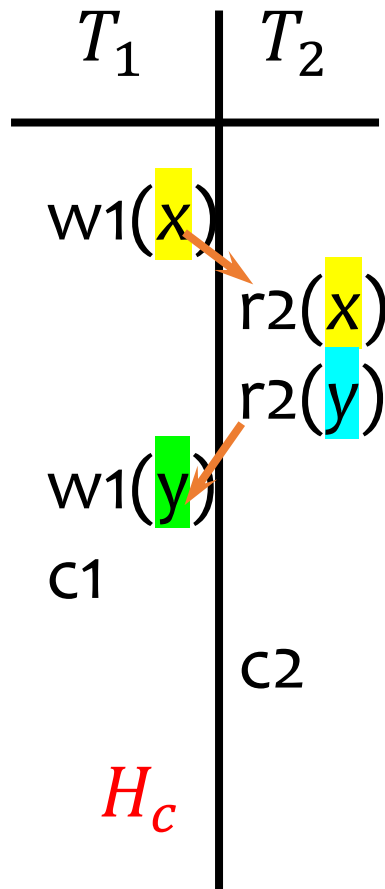
$w_1[x]$  and  $r_2[x]$  conflict, and  $w_1[x] < r_2[x]$   
 $w_1[y]$  and  $r_2[y]$  conflict, and  $w_1[y] < r_2[y]$

Serialization graph: no cycles  $\rightarrow$  serializable



# Example

- Example:  $H_c = w_1[x]r_2[x]r_2[y]w_1[y]c_1c_2$



$w_1[x]$  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: no conflicting pairs, as all are reads

- Related to y:  $w_4[y], r_1[y], w_3[y]$

- $w_4[y] < r_1[y]$   $T_4 \rightarrow T_1$

- $w_4[y] < w_3[y]$   $T_4 \rightarrow T_3$

- $r_1[y] < w_3[y]$   $T_1 \rightarrow T_3$

- Related to z:  $w_4[z], r_2[z], w_2[z], r_3[z], r_1[z]$

- $w_4[z] < r_2[z]$   $T_4 \rightarrow T_2$

- $w_4[z] < w_2[z]$   $T_4 \rightarrow T_2$

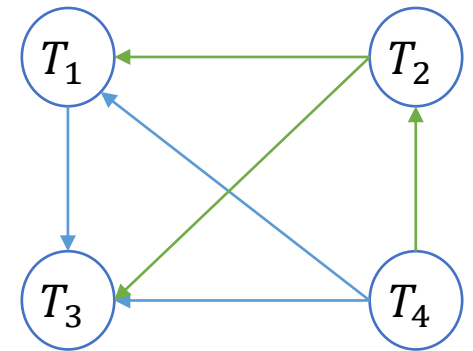
- $w_4[z] < r_3[z]$   $T_4 \rightarrow T_3$

- $w_4[z] < r_1[z]$   $T_4 \rightarrow T_1$

- $r_2[z], w_2[z]$  are not, as they are from the same transactions

- $w_2[z] < r_3[z]$   $T_2 \rightarrow T_3$

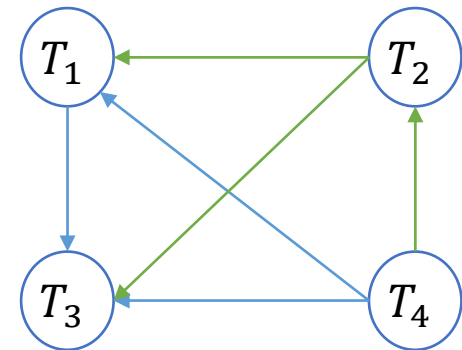
- $w_2[z] < r_1[z]$   $T_2 \rightarrow T_1$



# 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:  $T_4 \rightarrow T_2 \rightarrow T_1 \rightarrow T_3$



- 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

- Transactions
  - Properties: ACID
- Isolation
  - Different isolation levels
    - Based on allowed anomalies: dirty reads, non-repeatable reads, phantoms
    - Serializability (focus)
  - Which isolation level to choose for SQL transactions?


# SQL isolation levels

Isolation level/anomaly	Dirty reads	Non-repeatable reads	Phantoms
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Impossible	Possible	Possible
REPEATABLE READ	Impossible	Impossible	Possible
SERIALIZABLE	Impossible	Impossible	Impossible

- Syntax: At the beginning of a transaction,  
**SET TRANSACTION ISOLATION LEVEL** *isolation\_level*  
**[READ ONLY | READ WRITE];**
  - READ UNCOMMITTED can only be **READ ONLY**
    - Update/Insertion/deletion query cannot have READ UNCOMMITTED
- PostgreSQL defaults to **READ COMMITTED**

# The lowest isolation level to set?

- -- T1:  
UPDATE User  
SET pop = 0.99  
WHERE uid = 142;  
COMMIT;




Isolation level	Possible anomalies for T1
READ UNCOMMITTED	Dirty reads
READ COMMITTED	No unrepeatable reads
REPEATABLE READ	No phantoms
SERIALIZABLE	No

- Consider other possible concurrent transactions
  - Assume each table is an object
  - T1 reads User only once, i.e. read(User), write(User)
  - For example, another transaction T' is updating uid
  - Lowest isolation level: read committed

# The lowest isolation level to set?

- -- T2:  
SELECT AVG(pop)  
FROM User;  
COMMIT;




Isolation level	Possible anomalies for T2
READ UNCOMMITTED	Dirty reads
READ COMMITTED	No unrepeatable reads
REPEATABLE READ	No phantoms
SERIALIZABLE	No

- Consider other possible concurrent transactions
  - Assume each table is an object
  - T1 reads User only once, i.e., Read(User)
  - For example, another transaction T' is updating pop
  - Lowest isolation level: read committed



# The lowest isolation level to set?

- -- T3:  
SELECT AVG(pop)  
FROM User;  
  
SELECT MAX(pop)  
FROM User;  
COMMIT;



Isolation level	Possible anomalies for T3
READ UNCOMMITTED	Dirty reads
READ COMMITTED	Unrepeatable reads
REPEATABLE READ	Phantoms
SERIALIZABLE*	No

- Consider other possible concurrent transactions
  - Assume each table is an object
  - T1 reads User twice: READ(User), READ(User)
  - For example, another transaction T' is inserting/updating/deleting a row to User
  - Lowest isolation level: serializable

# Summary

- Transactions
  - Properties: ACID
- Isolation
  - Different isolation levels
  - Serializability (focus)
  - Which isolation level to choose for SQL transactions?

Isolation level/anomaly	Dirty reads	Non-repeatable reads	Phantoms
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Impossible	Possible	Possible
REPEATABLE READ	Impossible	Impossible	Possible
SERIALIZABLE	Impossible	Impossible	Impossible