Transactions and Deadlocks

CS 3200: Database Design

CS 5200: Database Systems

Transaction Support

- A **transaction** is a set of updates that must either succeed together or fail together. If only some of the updates were to succeed, the database could be left in an inconsistent state.

- If transaction fails, the database is **rolled back** to its original state.

- A transaction is a **logical unit of work** meaning that each transaction does something useful and no *part of a transaction achieves anything of use or of interest to the user. (Consistency may be violated during the transaction.)*

- Transactions are a unit of **recovery, consistency, and integrity**

Syntactic Elements

START TRANSACTION;

ROLLBACK; (Transaction Failed)

- Rollback signals the **unsuccessful** end of a transaction
- *Returns the system to the state it was in before the transaction began*
- System state must be the same as if the transaction had never existed
- Must abort any transactions that depend on the outcome of the aborting transaction

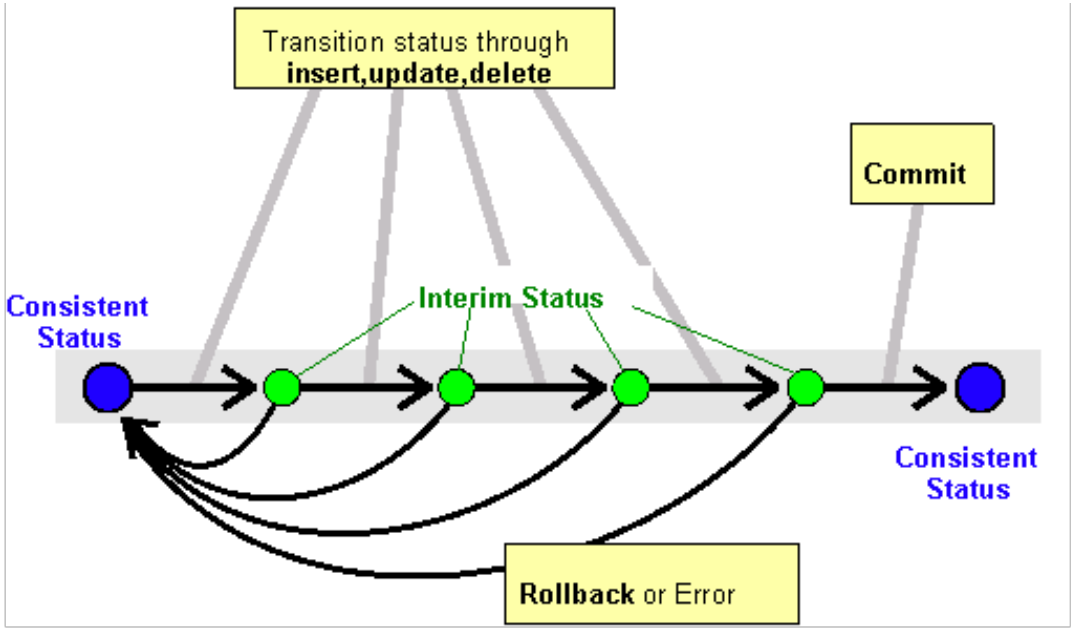
COMMIT; (Transaction Succeeded)

- COMMIT signals the **successful** end of a transaction. Any changes made by the transaction should be saved. These changes are now visible to other transactions
- Declare the transaction permanently complete
- If you commit, no actions should be able to move the DBMS to a state not containing the results of the transaction. All operations must be forever persistent in the database.
- Committed transactions cannot be rolled back

Transaction Examples

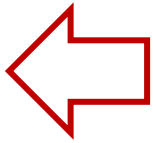
- Transferring money from one account to another
- Receiving money from an ATM machine
- Inserting orders and associated line items
- On-line purchase: charge credit card, reserve inventory, initiate shipping
- Etc.

All of these examples involve intermediate states where the database is temporarily inconsistent.

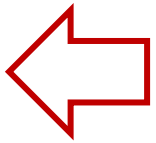


An example stored procedure with a transaction

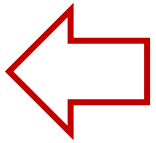
```
-- A stored procedure with a transaction
DELIMITER //

CREATE PROCEDURE test()
BEGIN
  DECLARE sql_error TINYINT DEFAULT FALSE;

  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
    SET sql_error = TRUE;

  START TRANSACTION;

  INSERT INTO invoices
  VALUES (115, 34, 'ZXA-080', '2015-01-18',
          14092.59, 0, 0, 3, '2015-04-18', NULL);

  INSERT INTO invoice_line_items
  VALUES (115, 1, 160, 4447.23, 'HW upgrade');

  INSERT INTO invoice_line_items
  VALUES (115, 2, 167, 9645.36, 'OS upgrade');

  IF sql_error = FALSE THEN
    COMMIT;
    SELECT 'The transaction was committed.';
  ELSE
    ROLLBACK;
    SELECT 'The transaction was rolled back.';
  END IF;

END//
```



Set a flag if an error occurs



Start the transaction – everything after this gets rolled back if a problem occurs



Rollback or Commit depending on whether an error occurred.

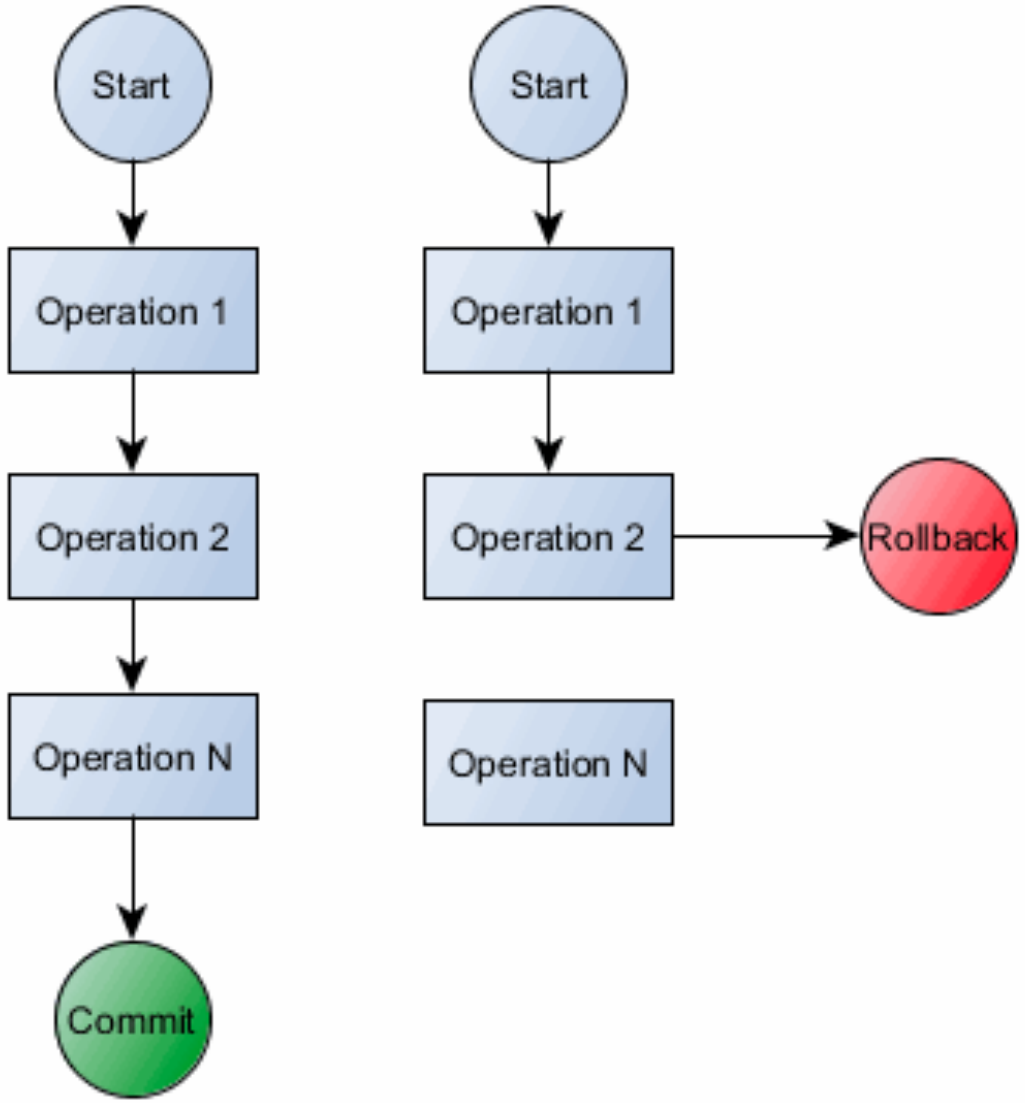
ACID Properties

Atomicity: Transactions are *all or nothing*: They either succeed or fail as a whole.

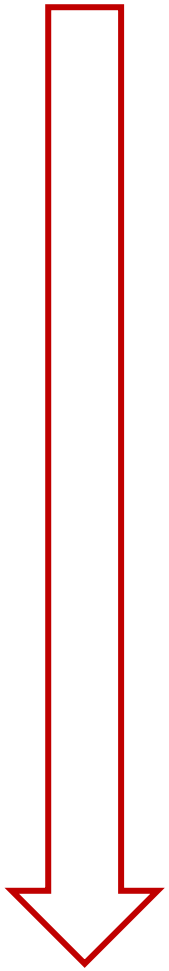
Consistency: Transactions on a database must transform the database from one consistent state to another.

Isolation: Partial effects of incomplete transactions should not be visible to other transactions.

Durability: Effects of a committed transaction are permanent and must not be lost because of a later failure.



Transfer $50 from account A to account B



Read (A)

A = A – 50

Write (A)

Read (B)

B = B + 50

Write (B)

Atomicity: Don’t take money from A without giving to B

Consistency: No money should be lost or gained

Isolation: Other queries shouldn’t see A or B change until the entire transaction is completed and both values are updated.

Durability: The money does not go back to A

Recovery Services

- Database hardware can fail: Servers, disk drives, and software can crash.
- Rollbacks need to occur in the event of these types of failure.
- Backups are your friend.

```
$ mysqldump --user root –p --databases myprojectdb > myproject_09JUN2020.sql
```

Transaction and Concurrency support are closely related

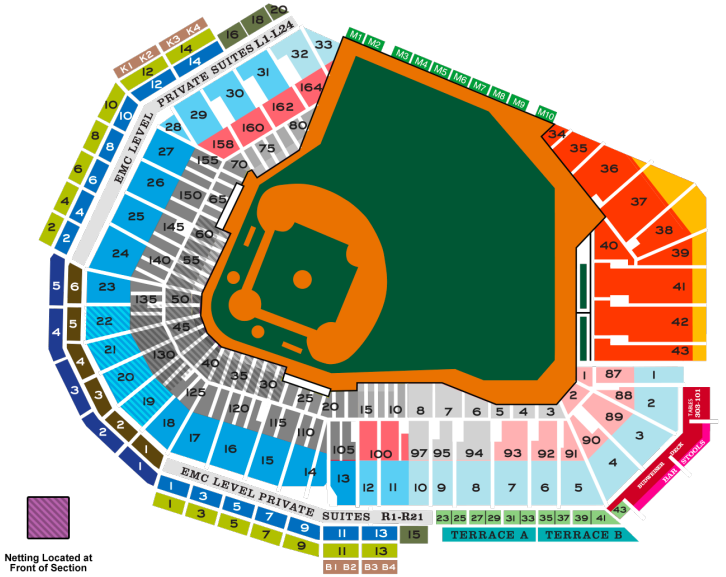
Concurrency Control is concerned with managing simultaneous operations on the database without having them interfere with one another.

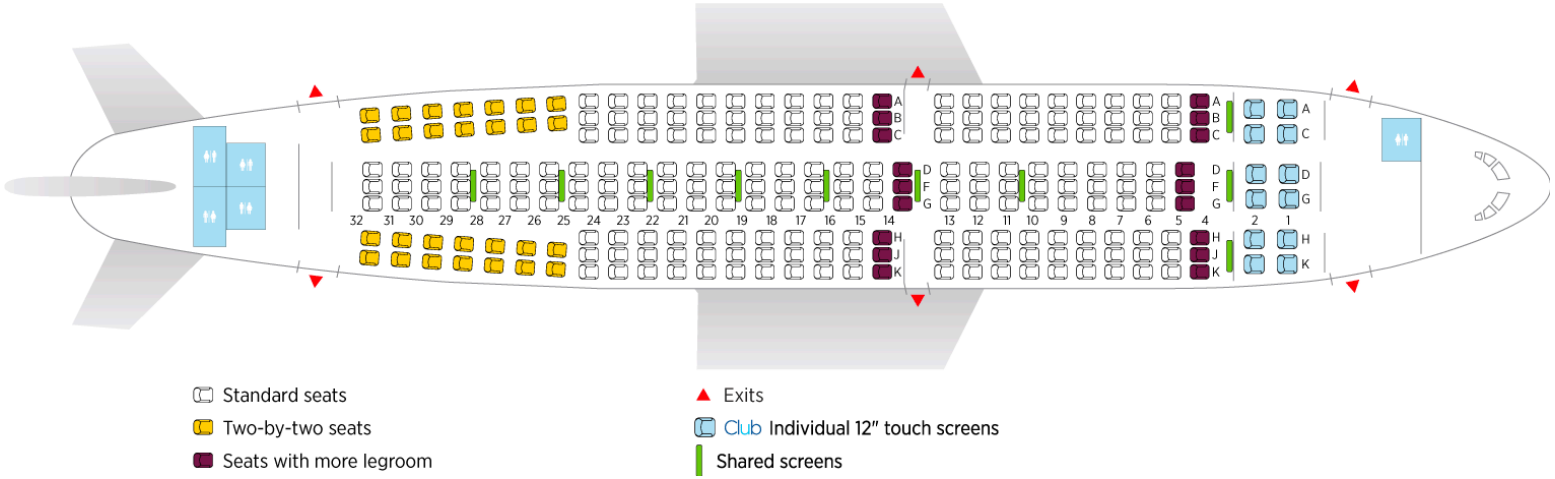
- Prevents interference when two or more users are accessing the database simultaneously and at least one is updating data.

- Although two transactions may be correct by themselves, interleaving the operations may produce an incorrect result.

Where do concurrency issues arise? Everywhere!

- Booking a flight or a hotel room
- Reserving a seat at the movie theater
- Accessing a shared account or a shared document
- Updating customer information
- Buying tickets to a concert or sporting event
- Reserving your brand new iPhone X
- Buying something online with limited inventory





Concurrency Support: Lost Updates

The *lost update* problem: Two updates occurring at the same time, but only the last update is saved. For example, multiple users sharing a single bank account doing deposits and withdrawals at different ATMs at the same time.

Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	read(bal_x)	100
t_3	read(bal_x)	$bal_x = bal_x + 100$	100
t_4	$bal_x = bal_x - 10$	write(bal_x)	200
t_5	write(bal_x)	commit	90
t_6	commit		90

Solution:
Loss of T_2's update avoided by preventing T_1 from reading bal_x until after update.

Concurrency Support: Uncommitted Dependencies

Uncommitted dependencies (aka *Dirty Reads*) occur when the first transaction updates a data object and second transaction reads that updated value BEFORE the first transaction commits. (2nd Transaction has access to uncommitted data.)

Time	T_3	T_4	bal_x
t_1		begin_transaction	100
t_2		read(bal_x)	100
t_3		$bal_x = bal_x + 100$	100
t_4	begin_transaction	write(bal_x)	200
t_5	read(bal_x)	⋮	200
t_6	$bal_x = bal_x - 10$	rollback	100
t_7	write(bal_x)		190
t_8	commit		190

Solution:
Prevent T_3 from reading bal_x until after T_4 commits or aborts.

Concurrency Support: Inconsistent Analysis

Inconsistent analysis (aka *Non-repeatable Reads / phantom reads*) occur when one transaction is reading a data object and still processing (e.g., summing the values in a column, or repeatedly accessing a particular value) while a second transaction is making updates to those very same values or inserting/deleting rows.

Time	T_5	T_6	bal_x	bal_y	bal_z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum = 0	100	50	25	0
t_3	read(bal_x)	read(bal_x)	100	50	25	0
t_4	$bal_x = bal_x - 10$	sum = sum + bal_x	100	50	25	100
t_5	write(bal_x)	read(bal_y)	90	50	25	100
t_6	read(bal_z)	sum = sum + bal_y	90	50	25	150
t_7	$bal_z = bal_z + 10$		90	50	25	150
t_8	write(bal_z)		90	50	35	150
t_9	commit	read(bal_z)	90	50	35	150
t_{10}		sum = sum + bal_z	90	50	35	185
t_{11}		commit	90	50	35	185

Solution:
Prevent T_6 from reading bal_x or bal_z until after T_5 has completed its updates.

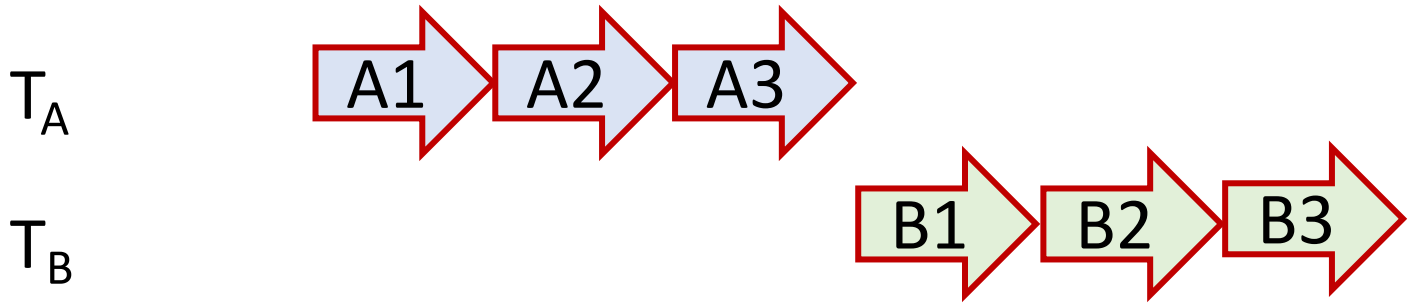
Transaction scheduling

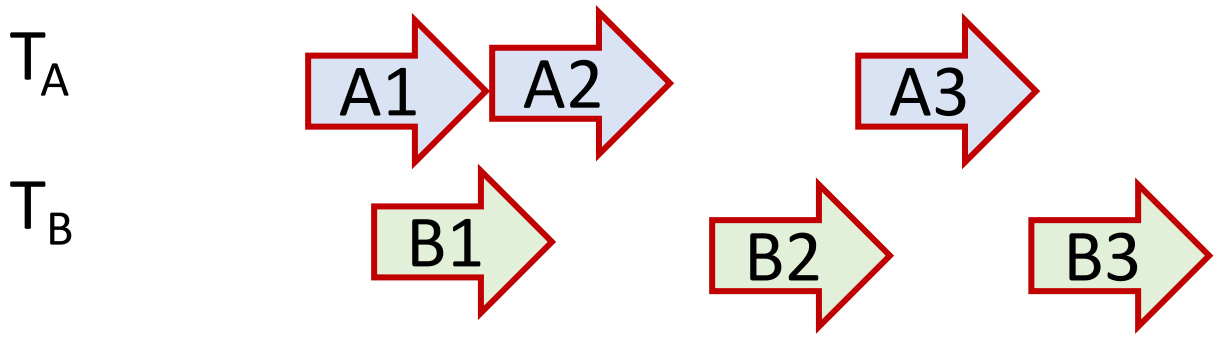
A **transaction schedule** defines the sequence of reads and write operations by a set of transactions. In any schedule, *the relative order of operations within the transaction must be maintained*.

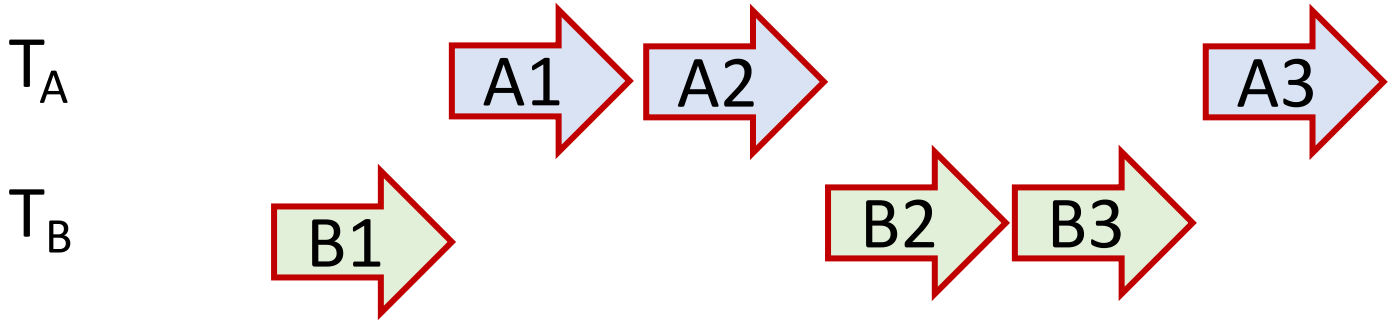
The goals are:

- Maintain database consistency
- Minimize run-time

Example schedules

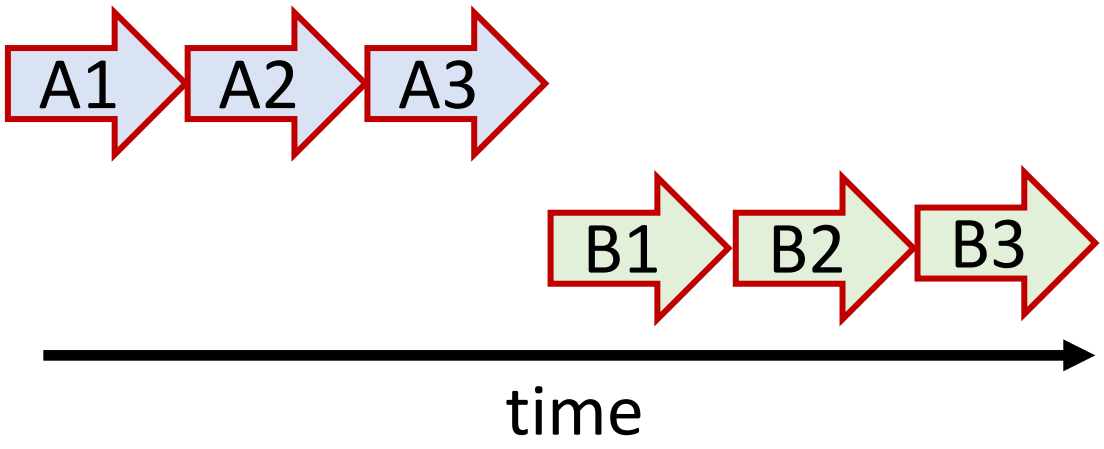






Simple scheduling: Serialization

- A **serial schedule** is a transaction schedule where operations of each transaction are executed consecutively without any interleaved operations from other transactions (each transaction commits before the next one is allowed to begin.)



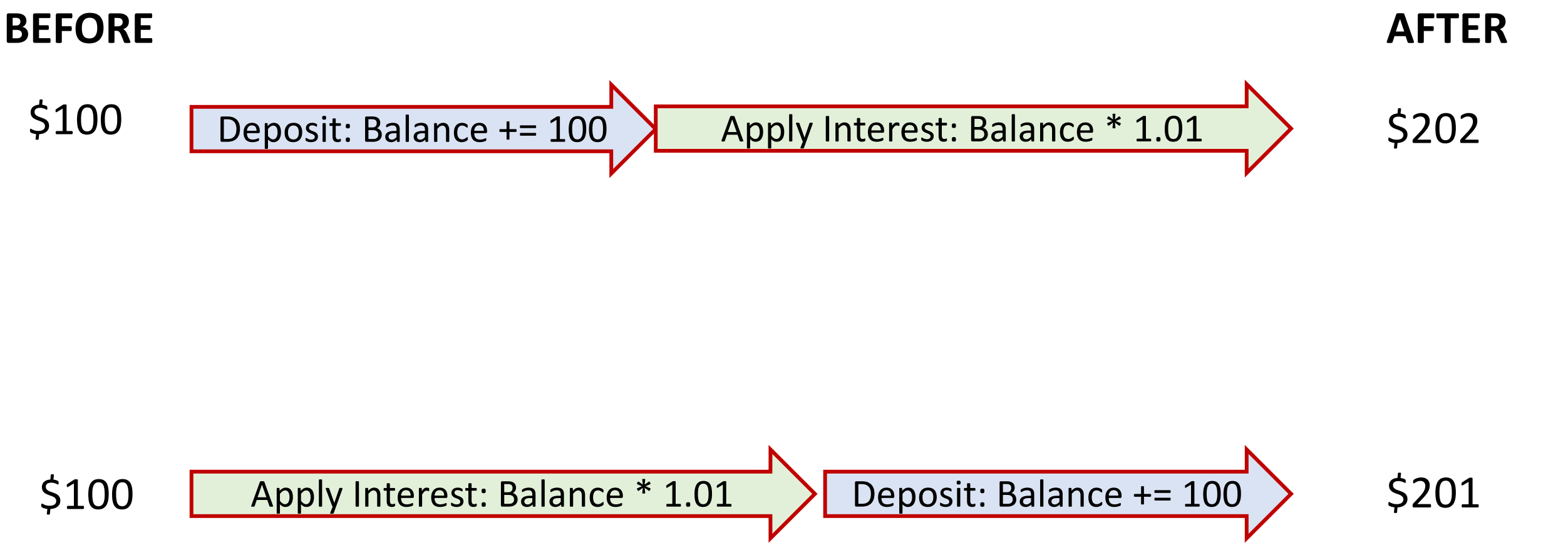
Advantages:

- Simple
- Guaranteed to give consistent results

Disadvantages:

- Cannot overlap I/O operations and computation
- Multiple cores or processors are sitting idle
- Response times get long as well as variable
- Short transactions must wait for long ones to finish

Serial schedules are not all equivalent



…but they all produce valid outcomes

General Observations

Transactions can safely run concurrently if:

- They only perform read operations.
- They are operating on different objects

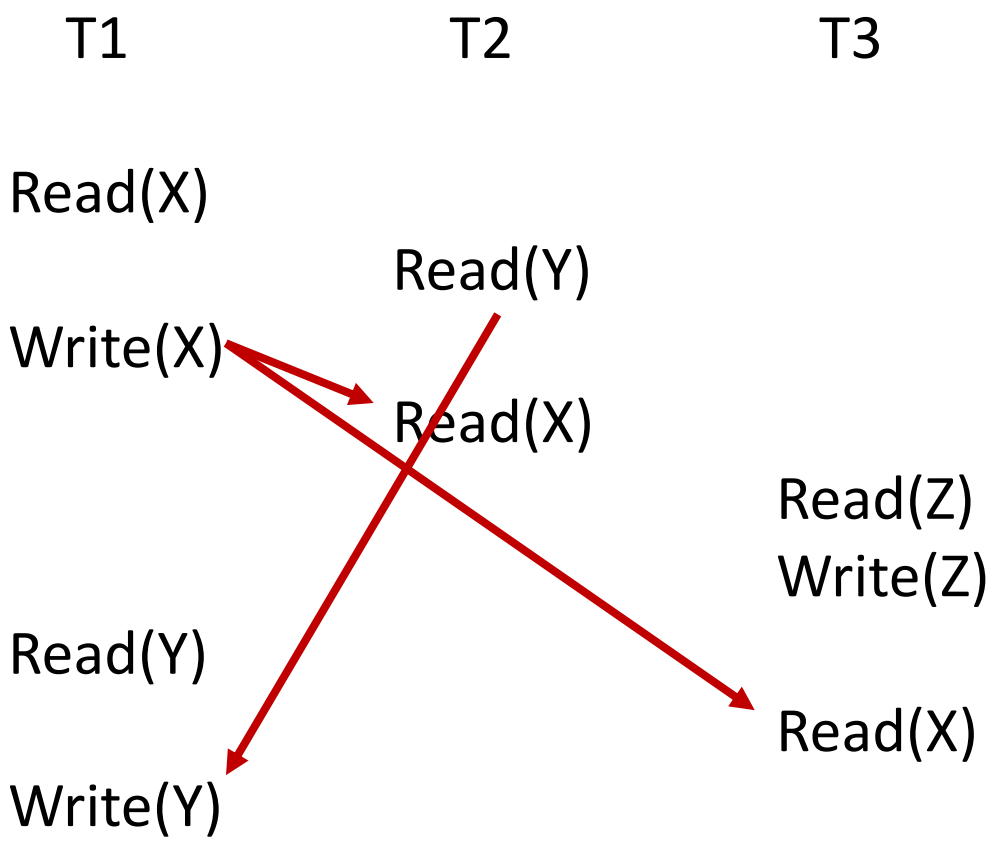
Transactions may conflict with each other and produce inconsistent results when:

- They are accessing the same data object, and
- one transaction writes a data item and another transaction reads or writes that **same data item.**

Conflicting Operations

Two operations are **conflicting** if

a) They belong to two different transactions
b) They operate on the same piece of data
c) At least one of the operations is a write



Non-serial schedules

- A **non-serial schedule** interleaves operations from set of transactions, allowing them to run concurrently without interference.
- A non-serial schedule is called **serializable** if it is *equivalent* to some serial schedule, i.e., it produces the same result as some serial execution.
- A schedule is **conflict serializable** if transactions in the schedule have a conflict but the schedule is still serializable.

Examples of Serializability: Equivalent Outcomes!

Non-serial

Time	T_7	T_8
t_1	begin_transaction	
t_2	read($\mathbf{bal_x}$)	
t_3	write($\mathbf{bal_x}$)	
t_4		begin_transaction
t_5		read($\mathbf{bal_x}$)
t_6		write($\mathbf{bal_x}$)
t_7	read($\mathbf{bal_y}$)	
t_8	write($\mathbf{bal_y}$)	
t_9	commit	
t_{10}		read($\mathbf{bal_y}$)
t_{11}		write($\mathbf{bal_y}$)
t_{12}		commit

(a)

Non-serial

T_7	T_8
begin_transaction	
read($\mathbf{bal_x}$)	
write($\mathbf{bal_x}$)	
	begin_transaction
	read($\mathbf{bal_x}$)
read($\mathbf{bal_y}$)	
	write($\mathbf{bal_x}$)
write($\mathbf{bal_y}$)	
commit	
	read($\mathbf{bal_y}$)
	write($\mathbf{bal_y}$)
	commit

(b)

Serial

T_7	T_8
begin_transaction	
read($\mathbf{bal_x}$)	
write($\mathbf{bal_x}$)	
read($\mathbf{bal_y}$)	
write($\mathbf{bal_y}$)	
commit	
	begin_transaction
	read($\mathbf{bal_x}$)
	write($\mathbf{bal_x}$)
	read($\mathbf{bal_y}$)
	write($\mathbf{bal_y}$)
	commit

(c)

Is a schedule conflict serializable?

Question: How do we know if a non-serial schedule is conflict serializable? In other words, if we have a non-serial schedule with conflicts, how do we know if it is equivalent to *some* serial schedule?

Answer: We create a *precedence graph.* If the precedence graph contains a cycle then it is **not** conflict serializable.

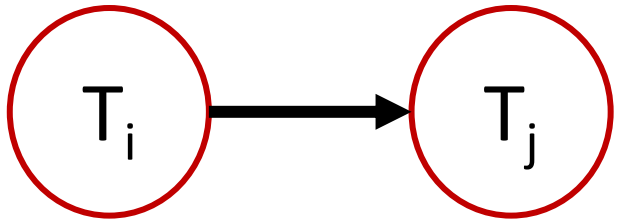
Precedence Graphs

A **precedence** graph is a directed graph that identifies dependencies among different concurrent transactions.

Each transaction is a **node**.

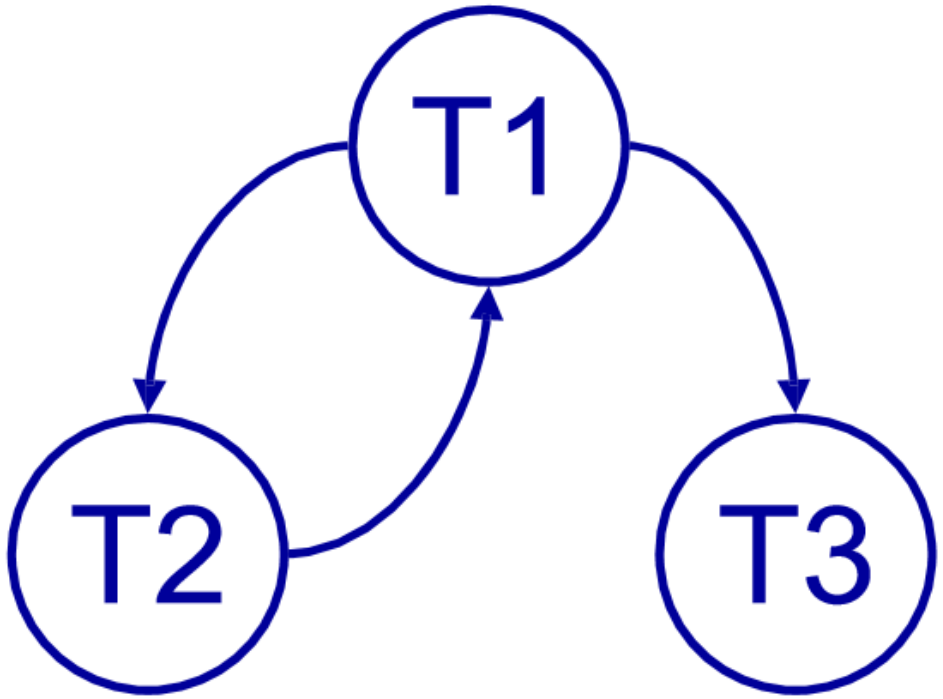
We add a **directed edge** $T_i \rightarrow T_j$ if:

- T_j reads the value of an item after it has been written by T_i
- T_j writes a value into an item after it has been read by T_i
- T_j writes a value into an item after it has been written by T_i



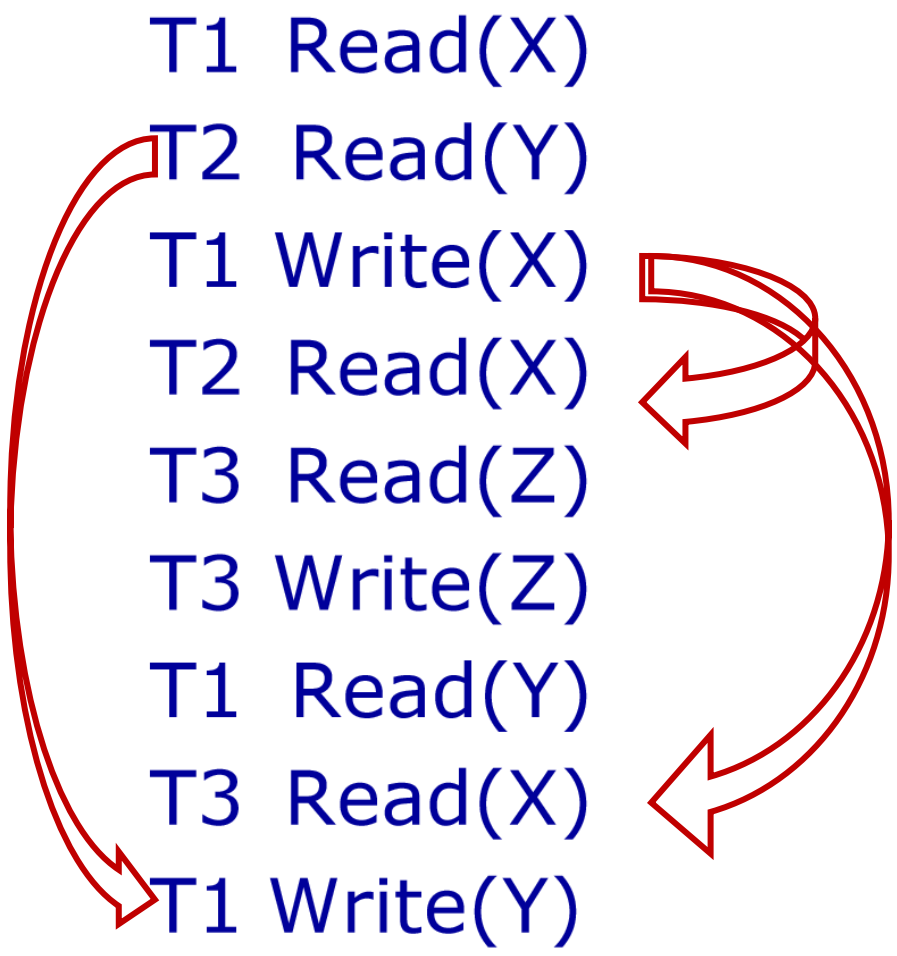
Example of a precedence graph with solution

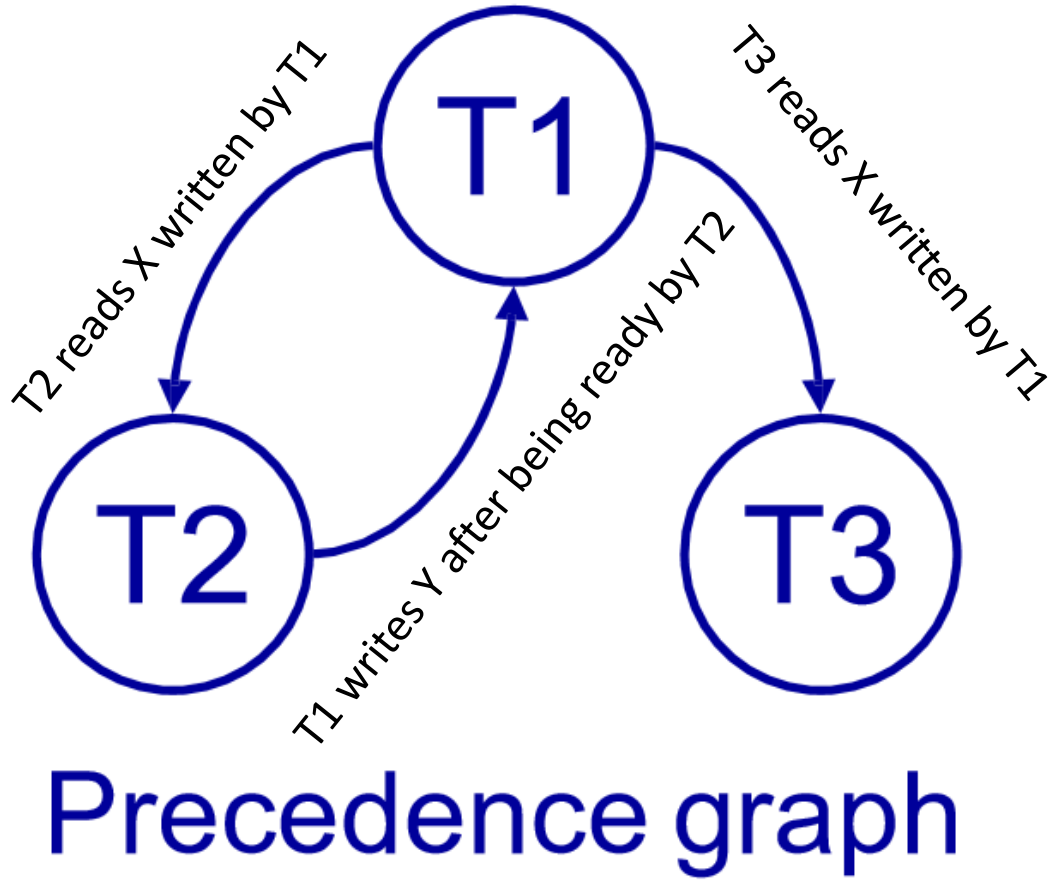
T1 Read(X)
T2 Read(Y)
T1 Write(X)
T2 Read(X)
T3 Read(Z)
T3 Write(Z)
T1 Read(Y)
T3 Read(X)
T1 Write(Y)



Precedence graph

Example of a precedence graph with solution





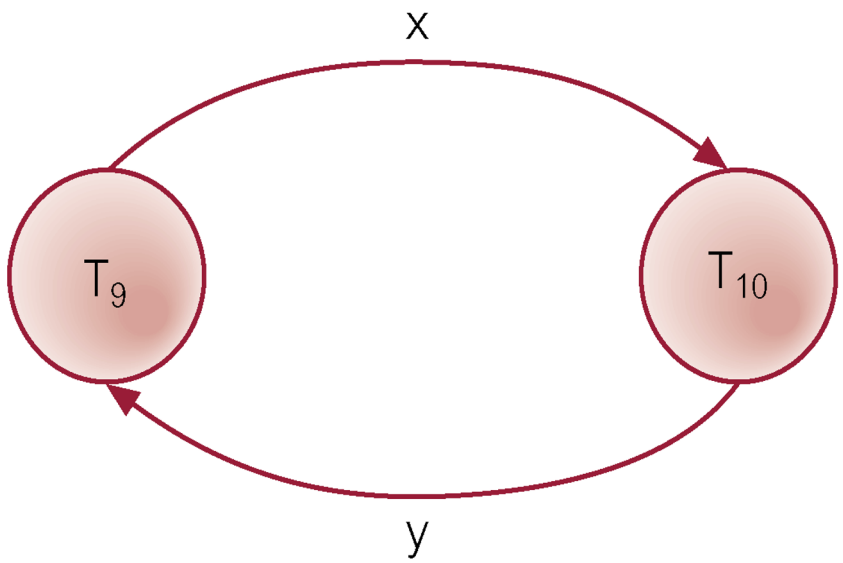
Precedence graph

Not Conflict-Serializable

Time			T_9	T_{10}
t_1			begin_transaction	
t_2	$200	$200	read(bal_x)	
t_3			$bal_x = bal_x + 100$	
t_4	$300	$200	write(bal_x)	begin_transaction
t_5				read(bal_x)
t_6				$bal_x = bal_x * 1.1$
t_7	$330	$200		write(bal_x)
t_8				read(bal_y)
t_9	$330	$220		$bal_y = bal_y * 1.1$
t_{10}				write(bal_y)
t_{11}			read(bal_y)	commit
t_{12}			$bal_y = bal_y - 100$	
t_{13}	$330	$120	write(bal_y)	
t_{14}			commit	

T_9 is transferring £100 from one account with balance bal_x to another account with balance bal_y.

T_{10} is increasing balance of these two accounts by 10%.



Locking

Transactions use **locks** to deny access to other transactions and so prevent incorrect updates.

Most widely used approach to ensure serializability.

Two types of locks:

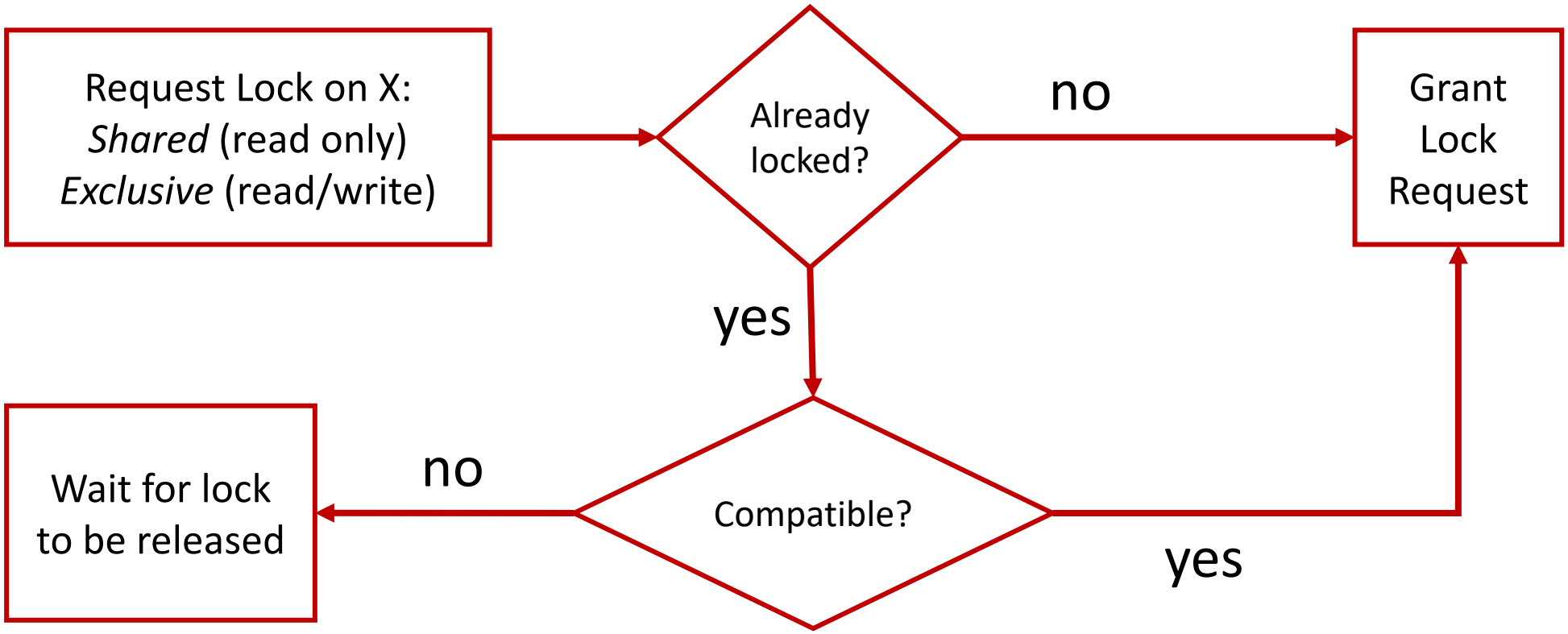
- **shared (read) lock:** A transaction can read the data item, but not update it.
- **exclusive (write) lock:** A transaction can both read and write the data item.

Lock *granularity*: how fine-grained the locking is

- Database
- Table
- Row
- Column
- Field

} Common, supported by InnoDB

Locking process: Transaction T accesses item X



(Requesting shared access to shared lock)

A valid locking schedule still doesn't fix the problem

Time			T_9	T_{10}
t_1			begin_transaction	
t_2	$200	$200	read(bal_x)	
t_3			$bal_x = bal_x + 100$	
t_4	$300	$200	write(bal_x)	begin_transaction
t_5				read(bal_x)
t_6				$bal_x = bal_x * 1.1$
t_7	$330	$200		write(bal_x)
t_8				read(bal_y)
t_9				$bal_y = bal_y * 1.1$
t_{10}	$330	$220		write(bal_y)
t_{11}			read(bal_y)	commit
t_{12}			$bal_y = bal_y - 100$	
t_{13}	$330	$120	write(bal_y)	
t_{14}			commit	

S = {

write_lock(T_9, bal_x)

read(T_9, bal_x), write(T_9, bal_x)

unlock(T_9, bal_x)

write_lock(T_{10}, bal_x)

read(T_{10}, bal_x), write(T_{10}, bal_x)

unlock(T_{10}, bal_x)

write_lock(T_{10}, bal_y)

read(T_{10}, bal_y), write(T_{10}, bal_y)

unlock(T_{10}, bal_y)

commit(T_{10}),

write_lock(T_9, bal_y)

read(T_9, bal_y), write(T_9, bal_y)

unlock(T_9, bal_y),

commit(T_9)

}

What went wrong?

- Transaction released its locks too soon, resulting in loss of total isolation and atomicity.
- To guarantee serializability, we need an additional protocol concerning the positioning of lock and unlock operations in every transaction.

S = {

write_lock(T_9, bal_x)

read(T_9, bal_x), write(T_9, bal_x)

unlock(T_9, bal_x)

write_lock(T_{10}, bal_x)

read(T_{10}, bal_x), write(T_{10}, bal_x)

unlock(T_{10}, bal_x)

write_lock(T_{10}, bal_y)

read(T_{10}, bal_y), write(T_{10}, bal_y)

unlock(T_{10}, bal_y)

commit(T_{10}),

write_lock(T_9, bal_y)

read(T_9, bal_y), write(T_9, bal_y)

unlock(T_9, bal_y),

commit(T_9)

}

2PL: Two-Phase Locking (not 2PL Compliant!)

- A Transaction follows the two-phase locking protocol if all locking operations (read_lock / write_lock) precede the first unlock operation in the transaction.
- Each transaction has a **growing phase** when locks are being acquired and a **shrinking phase** when locks are being released.
- Locks need not be acquired simultaneously but no lock is released until no new locks are required.

S = {

write_lock(T_9, bal_x)

read(T_9, bal_x), write(T_9, bal_x)

unlock(T_9, bal_x) ⇦

write_lock(T_{10}, bal_x)

read(T_{10}, bal_x), write(T_{10}, bal_x)

⇨ unlock(T_{10}, bal_x)

✕ write_lock(T_{10}, bal_y)

read(T_{10}, bal_y), write(T_{10}, bal_y)

unlock(T_{10}, bal_y)

commit(T_{10}),

write_lock(T_9, bal_y) ✕

read(T_9, bal_y), write(T_9, bal_y)

unlock(T_9, bal_y),

commit(T_9)

}

Preventing lost updates with 2PL

Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	write_lock(bal_x)	100
t_3	write_lock(bal_x)	read(bal_x)	100
t_4	WAIT	$bal_x = bal_x + 100$	100
t_5	WAIT	write(bal_x)	200
t_6	WAIT	commit/unlock(bal_x)	200
t_7	read(bal_x)		200
t_8	$bal_x = bal_x - 10$		200
t_9	write(bal_x)		190
t_{10}	commit/unlock(bal_x)		190

Preventing uncommitted dependency problem with 2PL

Time	T_3	T_4	bal_x
t_1		begin_transaction	100
t_2		write_lock(bal_x)	100
t_3		read(bal_x)	100
t_4	begin_transaction	$bal_x = bal_x + 100$	100
t_5	write_lock(bal_x)	write(bal_x)	200
t_6	WAIT	rollback/unlock(bal_x)	100
t_7	read(bal_x)		100
t_8	$bal_x = bal_x - 10$		100
t_9	write(bal_x)		90
t_{10}	commit/unlock(bal_x)		90

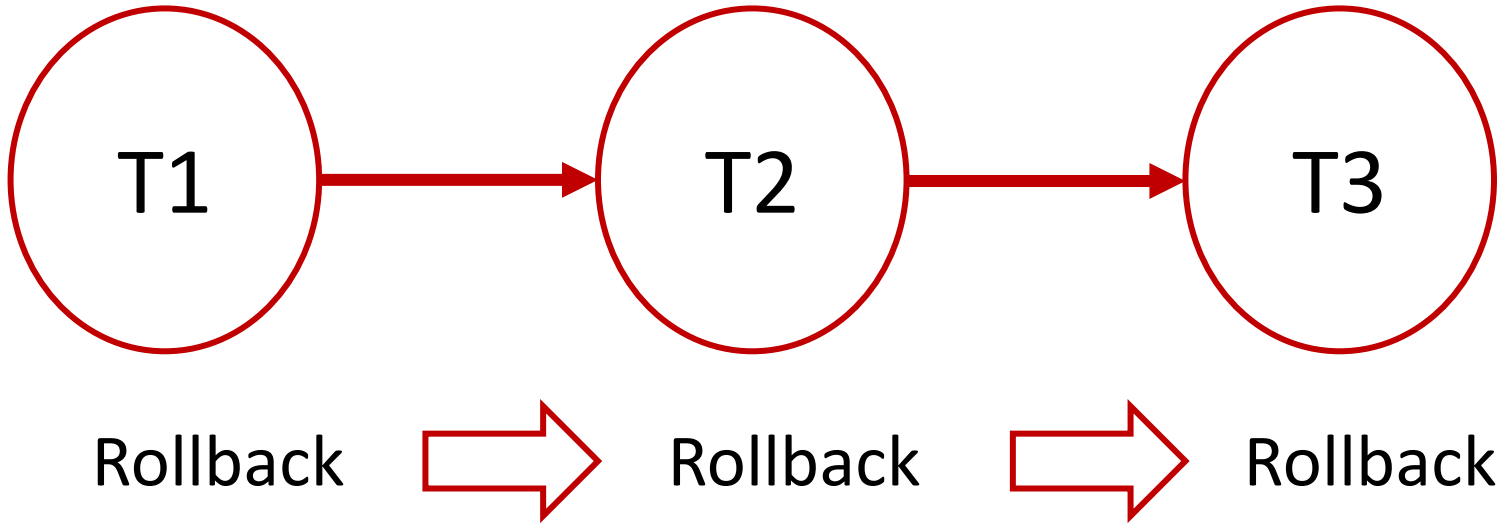
Preventing inconsistent analysis with 2PL

Time	T_5	T_6	bal_x	bal_y	bal_z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum = 0	100	50	25	0
t_3	write_lock(bal_x)		100	50	25	0
t_4	read(bal_x)	read_lock(bal_x)	100	50	25	0
t_5	$bal_x = bal_x - 10$	WAIT	100	50	25	0
t_6	write(bal_x)	WAIT	90	50	25	0
t_7	write_lock(bal_z)	WAIT	90	50	25	0
t_8	read(bal_z)	WAIT	90	50	25	0
t_9	$bal_z = bal_z + 10$	WAIT	90	50	25	0
t_{10}	write(bal_z)	WAIT	90	50	35	0
t_{11}	commit/unlock(bal_x, bal_z)	WAIT	90	50	35	0
t_{12}		read(bal_x)	90	50	35	0
t_{13}		sum = sum + bal_x	90	50	35	90
t_{14}		read_lock(bal_y)	90	50	35	90
t_{15}		read(bal_y)	90	50	35	90
t_{16}		sum = sum + bal_y	90	50	35	140
t_{17}		read_lock(bal_z)	90	50	35	140
t_{18}		read(bal_z)	90	50	35	140
t_{19}		sum = sum + bal_z	90	50	35	175
t_{20}		commit/unlock(bal_x, bal_y, bal_z)	90	50	35	175

Cascading rollback

- If *every* transaction in a schedule follows the two-phase locking protocol, then the schedule is guaranteed to be conflict serializable (*Eswaran et al.*, 1976)

- Problems can still occur such as cascading rollbacks:



An example of a cascading rollback

Time	T_{14}	T_{15}	T_{16}
t_1	begin_transaction		
t_2	write_lock($\mathbf{bal_x}$)		
t_3	read($\mathbf{bal_x}$)		
t_4	read_lock($\mathbf{bal_y}$)		
t_5	read($\mathbf{bal_y}$)		
t_6	$\mathbf{bal_x} = \mathbf{bal_y} + \mathbf{bal_x}$		
t_7	write($\mathbf{bal_x}$)		
t_8	unlock($\mathbf{bal_x}$)	begin_transaction	
t_9	⋮	write_lock($\mathbf{bal_x}$)	
t_{10}	⋮	read($\mathbf{bal_x}$)	
t_{11}	⋮	$\mathbf{bal_x} = \mathbf{bal_x} + 100$	
t_{12}	⋮	write($\mathbf{bal_x}$)	
t_{13}	⋮	unlock($\mathbf{bal_x}$)	
t_{14}	⋮	⋮	
t_{15}	rollback	⋮	
t_{16}		⋮	begin_transaction
t_{17}		⋮	read_lock($\mathbf{bal_x}$)
t_{18}		rollback	⋮
t_{19}			rollback

Preventing cascading rollback

Rigorous 2PL: Release *all* locks only at the end of the transaction.

Strict 2PL: Hold only *exclusive locks* until the end of the transaction.

Deadlocks

Deadlocks

An impasse that may result when two (or more) transactions are each waiting for locks held by the other to be released.

Time	T_{17}	T_{18}
t_1	begin_transaction	
t_2	write_lock($\mathbf{bal_x}$)	begin_transaction
t_3	read($\mathbf{bal_x}$)	write_lock($\mathbf{bal_y}$)
t_4	$\mathbf{bal_x} = \mathbf{bal_x} - 10$	read($\mathbf{bal_y}$)
t_5	write($\mathbf{bal_x}$)	$\mathbf{bal_y} = \mathbf{bal_y} + 100$
t_6	write_lock($\mathbf{bal_y}$)	write($\mathbf{bal_y}$)
t_7	WAIT	write_lock($\mathbf{bal_x}$)
t_8	WAIT	WAIT
t_9	WAIT	WAIT
t_{10}	⋮	WAIT
t_{11}	⋮	⋮

Deadlocks

Three general techniques for handling deadlock:

- Timeouts.
- Deadlock prevention.
- Deadlock detection and recovery.

Deadlocks - Timeout

- Transaction that requests lock will only wait for a system-defined period of time.
- If lock has not been granted within this period, lock request times out.
- In this case, DBMS assumes transaction may be deadlocked, even though it may not be, and it aborts and automatically restarts the transaction.

Deadlocks - deadlock prevention

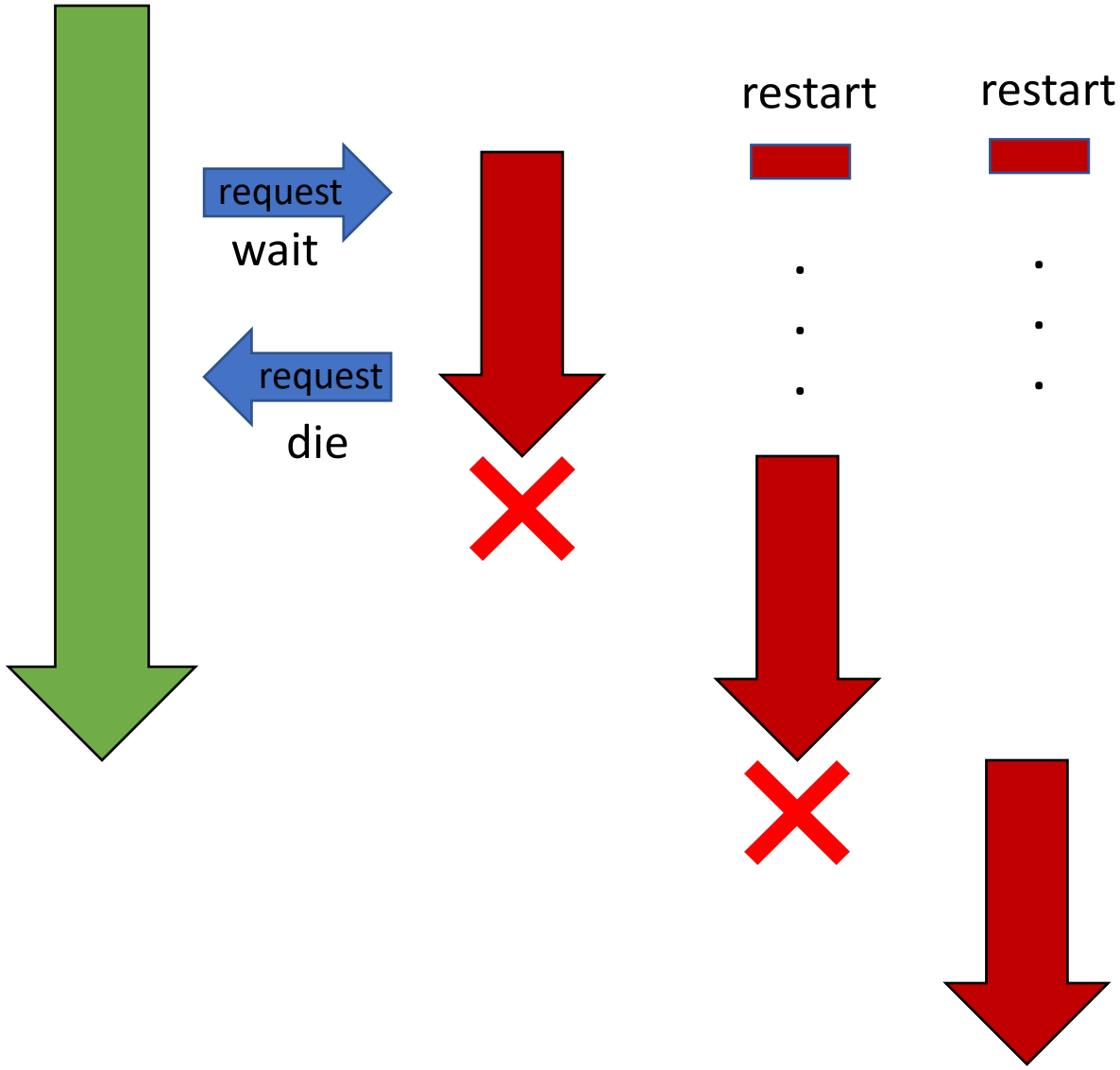
- DBMS looks ahead to see if transaction would cause deadlock and never allows deadlock to occur.

- Could order transactions using transaction timestamps:

  Wait-Die - only an older transaction can wait for younger one, otherwise transaction is aborted (*dies*) and restarted with same timestamp. (It eventually becomes the oldest and won't die.)

  Wound-Wait - only a younger transaction can wait for an older one. If older transaction requests lock held by younger one, younger one is aborted (*wounded*).

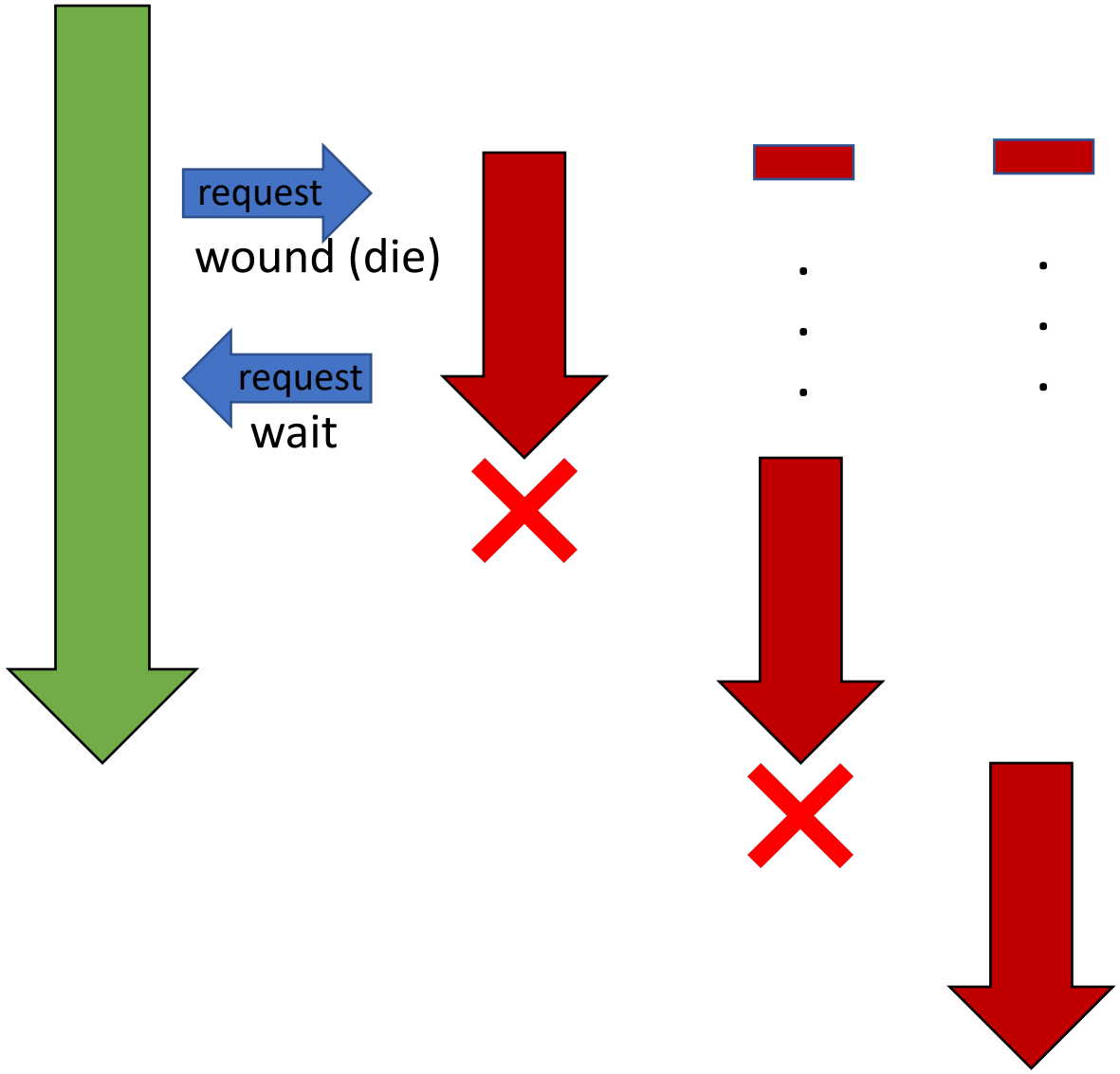
Wait-Die



Older transactions requesting resources held by younger transactions can wait.

Younger transactions requesting resources held by older transactions are killed and restarted.

Wound-Wait



Younger transactions requesting resources held by older transactions will wait.

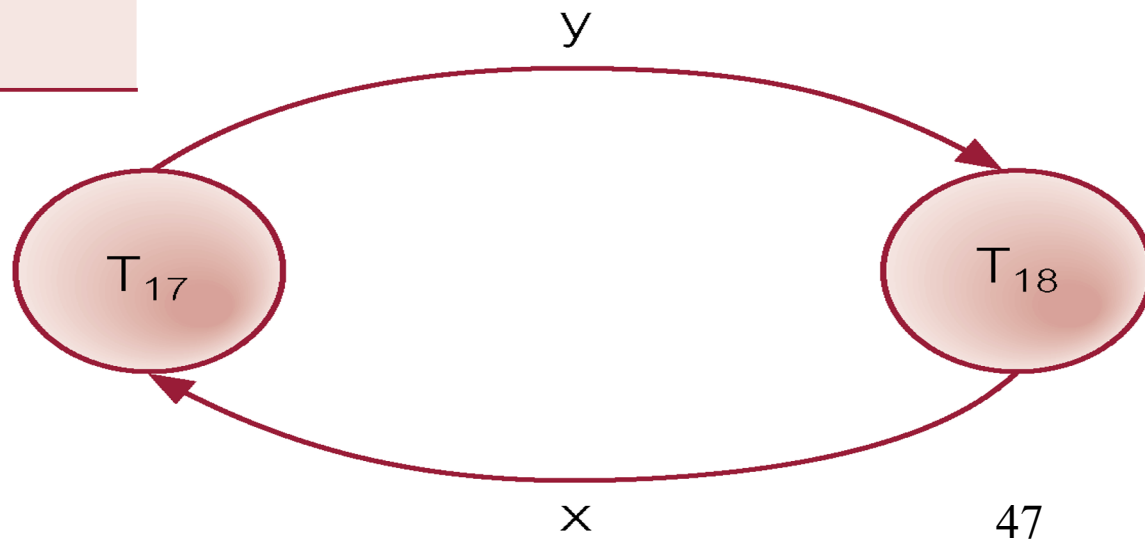
Older transactions requesting resources held by younger transactions force younger to abort and restart.

Deadlocks - deadlock detection and recovery

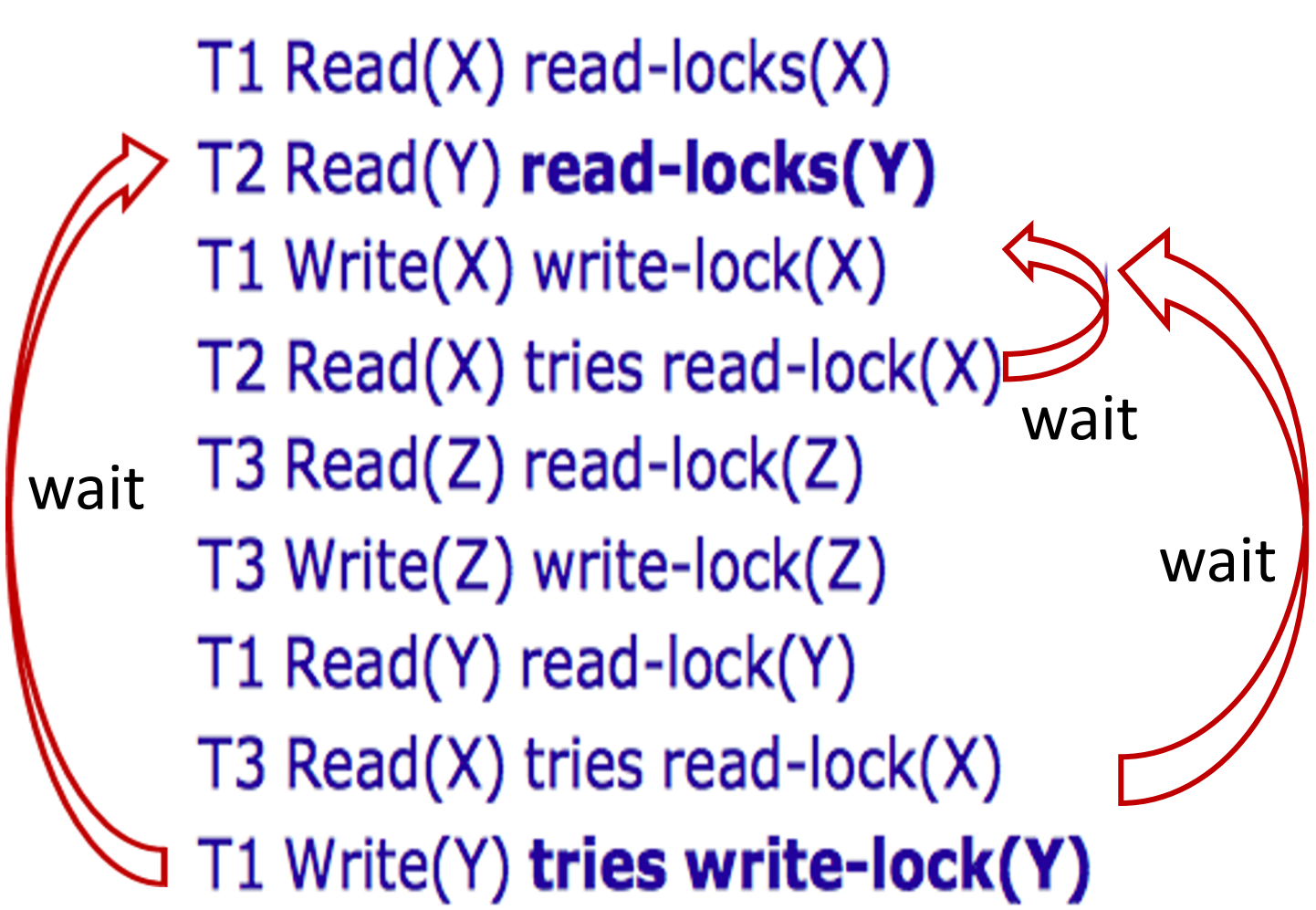
- DBMS allows deadlock to occur but recognizes it and breaks it.
- Usually handled by construction of wait-for graph (WFG) showing transaction dependencies:
  - Create a node for each transaction.
  - Create edge $T_i \rightarrow T_j$, if T_i is waiting for item locked by T_j.
- Deadlock exists if and only if WFG contains cycle.
- WFG is created at regular intervals.

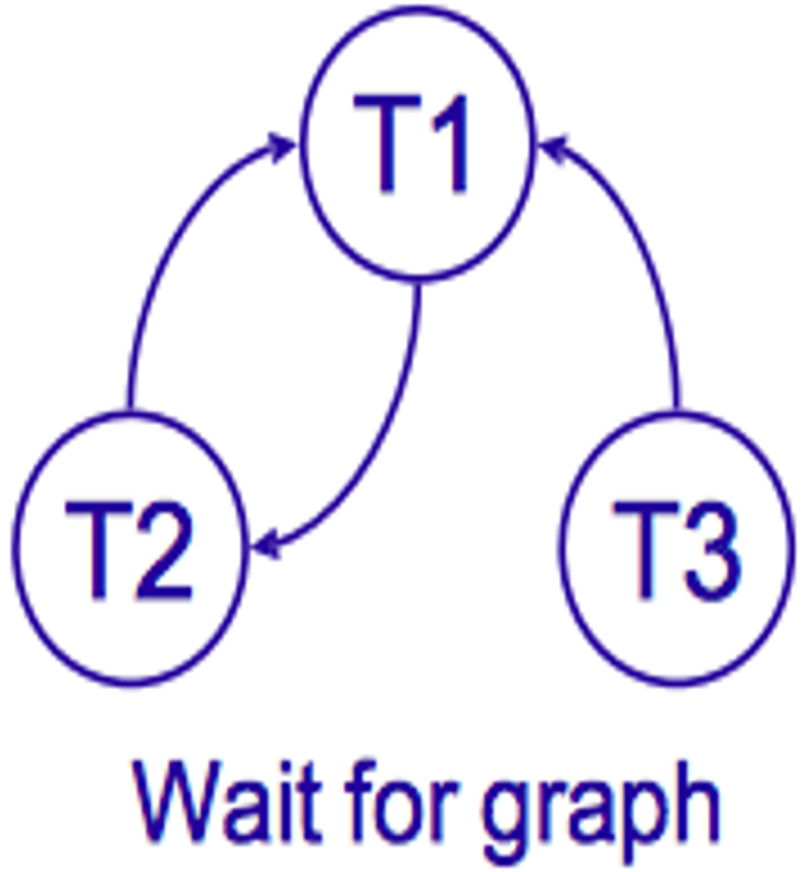
Example - Wait-For-Graph (WFG)

Time	T_{17}	T_{18}
t_1	begin_transaction	
t_2	write_lock($\mathbf{bal_x}$)	begin_transaction
t_3	read($\mathbf{bal_x}$)	write_lock($\mathbf{bal_y}$)
t_4	$\mathbf{bal_x} = \mathbf{bal_x} - 10$	read($\mathbf{bal_y}$)
t_5	write($\mathbf{bal_x}$)	$\mathbf{bal_y} = \mathbf{bal_y} + 100$
t_6	write_lock($\mathbf{bal_y}$)	write($\mathbf{bal_y}$)
t_7	WAIT	write_lock($\mathbf{bal_x}$)
t_8	WAIT	WAIT
t_9	WAIT	WAIT
t_{10}	⋮	WAIT
t_{11}	⋮	⋮



Wait for Graph example





Recovery from Deadlock Detection

- **choice of deadlock victim -** abort transaction that incur the min costs (running time, number of updates so far or still to occur).
- **how far to roll a transaction back -** may be possible to resolve deadlock by rolling back part of the transaction
- **avoiding starvation -** Not always picking the same transaction as a victim (keep a count of number of time a transaction is selected)