COURSE 2

Transactions and Concurrency Management

Scheduling Transactions

A *schedule* is a sequential order of the instructions

(*Read / Write / Abort / Commit*)

of *n* transactions such that the ordering of the instructions of each transaction is preserved

Scheduling Transactions

```
   T1:                    T2:
read(A)
read(sum)
                       read(A)
                       A := A + 20
                       write(A)
                       commit
read(A)
sum := sum + A
write(sum)
commit
```

Schedule

```
read1(A)
read1(sum)
read2(A)

write2(A)
commit2
read1(A)

write1(sum)
commit1
```

Scheduling Transactions

- *Serial schedule:* Schedule that does not interleave the actions of different transactions.

T1:	T2:
	read(A)
	A := A + 20
	write(A)
	commit
read(A)	
read(sum)	
read(A)	
sum := sum + A	
write(sum)	
commit	

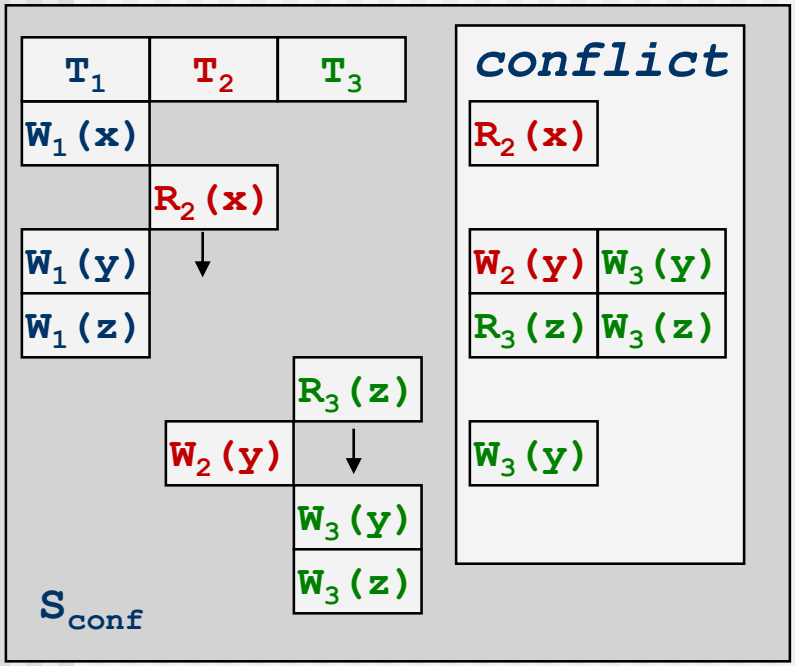
- *Non-serial schedule*: A schedule where the ops from a set of concurrent transactions are interleaved.

Scheduling Transactions

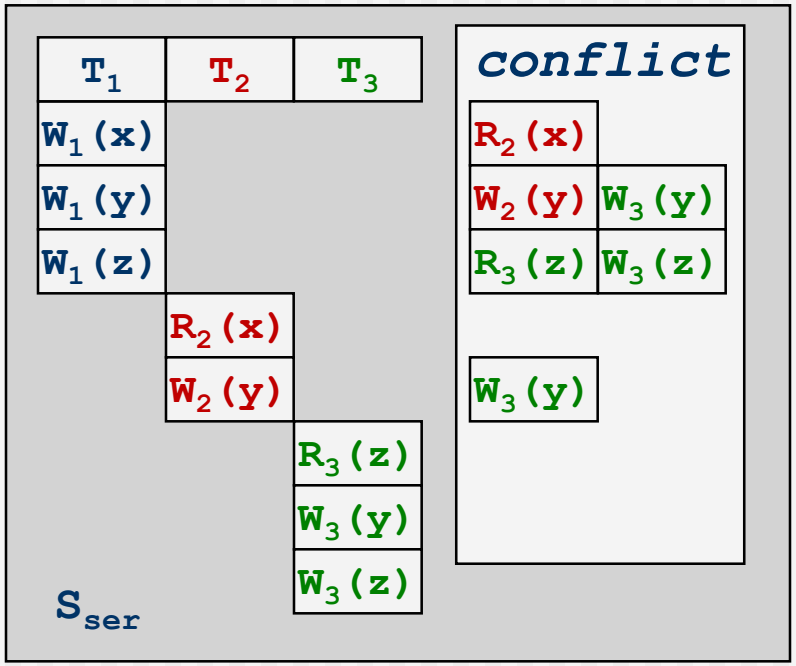
- *Equivalent schedules*: For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.
- *Serializable schedule*: A non-serial schedule that is equivalent to some serial execution of the transactions. (Note: If each transaction preserves consistency, every serializable schedule preserves consistency.)
- Checking serializability: which actions cannot be swapped in a schedule?:
  - Actions within the same transaction
  - Actions in different transitions transactions *on the same object* if at least one action is a **write** operation. (conflicting actions!)

Scheduling Transactions (cont)

- Two schedules are *conflict equivalent* if:
  - Involve the same actions of the same transactions
  - Every pair of conflicting actions is ordered the same way
- Schedule S is *conflict serializable* if S is conflict equivalent to some serial schedule



conflict-serializable



serial schedule

Serializability

- The objective of *serializability* is to find non-serial schedules that allow transactions to execute concurrently without interfering with one another, and thereby produce a database state that could be produced by a serial execution.
- It is important to guarantee serializability of concurrent transactions in order to prevent inconsistency from transactions interfering with one another.
- In serializability, the ordering of read and write operations is important.

Dependency Graph

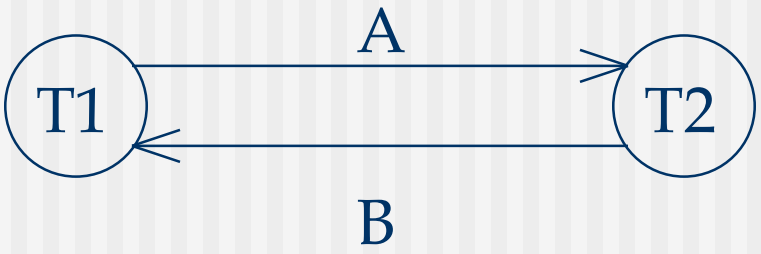
- *Dependency graph* (also called *serializability graph, precedence graph*):
  - One node per transaction
  - Edge from T_i to T_j if an action of *Ti* precedes and conflicts with one of *Tj*'s actions.

- Theorem: Schedule is conflict serializable if and only if its dependency graph is acyclic

Example

- A schedule that is not conflict serializable:

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



Dependency graph

- The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.

Algorithm for Testing Conflict Serializability of **S**

1. For each transaction T_i in **S** create a node labeled T_i in the precedence graph.
2. For each case in **S** where T_j executes a Read(x) after a Write(x) executed by T_i create an edge (T_i,T_j) in the precedence graph
3. For each case in **S** where T_j executes a Write(x) after a Read(x) executed by T_i create an edge (T_i,T_j) in the precedence graph
4. For each case in **S** where T_j executes a Write(x) after a Write(x) executed by T_i create an edge (T_i,T_j) in the precedence graph
5. **S** is conflict serializable iff the precedence graph has no cycles

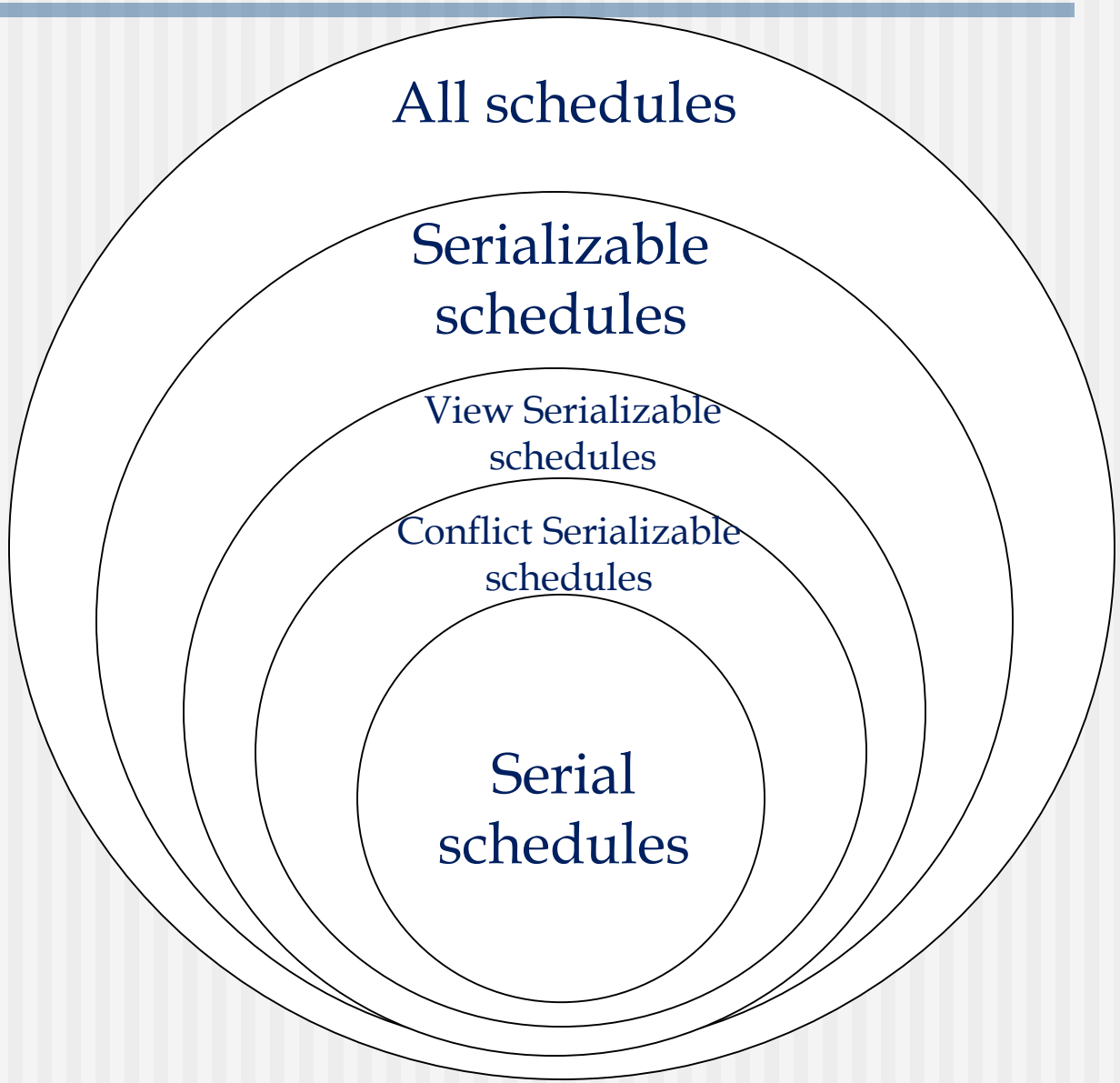
View Serializability

- Schedules S_1 and S_2 are view equivalent if:
  - If T_i reads initial value of A in S_1, then T_i also reads initial value of A in S_2
  - If T_i reads value of A written by T_j in S_1, then T_i also reads value of A written by T_j in S_2
  - If T_i writes final value of A in S_1, then T_i also writes final value of A in S_2

T1:	R(A)		W(A)
T2:		W(A)	
T3:			W(A)

T1:	R(A),W(A)		
T2:		W(A)	
T3:			W(A)

Transactions schedules



Another Serializable Schedule

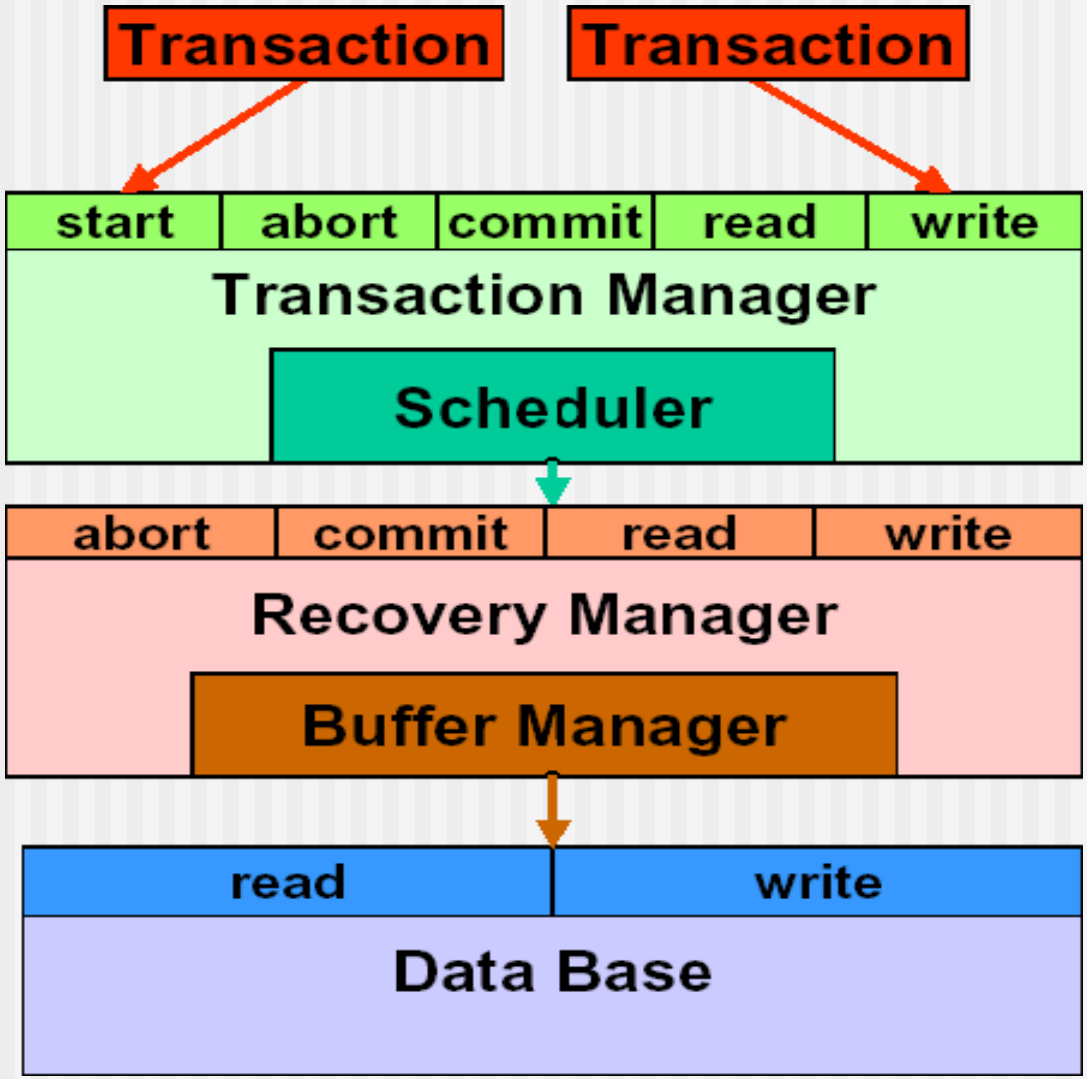
- The following schedule is serializable but it is not conflict serializable or view serializable

T1:	T2:
read(A)	
A := A-50	
write(A)	
	read(B)
	B := B - 10
	write(B)
read(B)	
B := B + 50	
write(B)	
	read(A)
	A := A + 10
	write(A)

Serializability in Practice

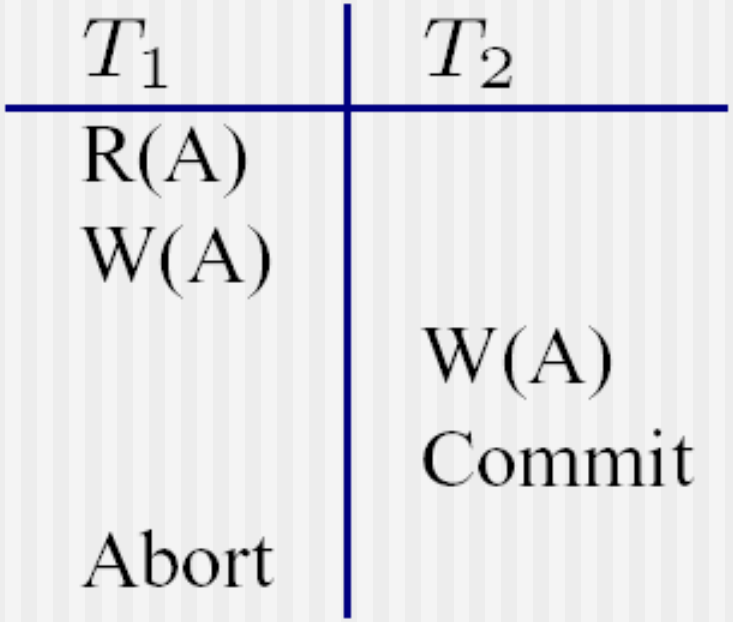
- In practice, a DBMS does not test for serializability of a given schedule. This would be impractical since the interleaving of operations from concurrent transactions could be dictated by the OS and thus could be difficult to impose.
- The approach taken by the DBMS is to use specific protocols that are known to produce serializable schedules.
- These protocols could reduce the concurrency but eliminate conflicting cases.

Transaction Execution



Recoverable Schedules

- In a *recoverable schedule* transactions **can only read** data that has been **already committed**
- There is still the situation of a **blind write**



- What should the value of A be after the abort?

Phantom Reads

- A transaction re-executes a query and finds that another committed transaction has inserted additional rows that satisfy the condition
  - If the rows have been modified or deleted, it is called an unrepeatable read

Example:

- T_1 executes `select * from Students where age < 25`
- T_2 executes `insert into Students values(12,'Jim', 23, 7)`
- T_2 commits
- T_1 executes `select * from Students where age < 25`

Lock-Based Concurrency Control

- We use locks to guarantee recoverable /serializable schedules
- A *locking protocol* is a set of rules to be followed by each transaction (enforced by the DBMS) to ensure that, even though actions of several transactions might be **interleaved**, the net effect is executing those transactions in **some** serial order.
- We will use *shared* and *exclusive* locks

Definitions

- **Locking**: A procedure used to control concurrent access to data. When one transaction is accessing the database, a lock may deny access to other transactions to prevent incorrect results.
- **Shared Lock** (or *read lock*): If a transaction has a shared lock on a data object, it can read the object but not update it.
- **Exclusive Lock** (or *write lock*): if a transaction has an exclusive lock on a data object, it can both read and update the object.

Locking-Based Algorithms

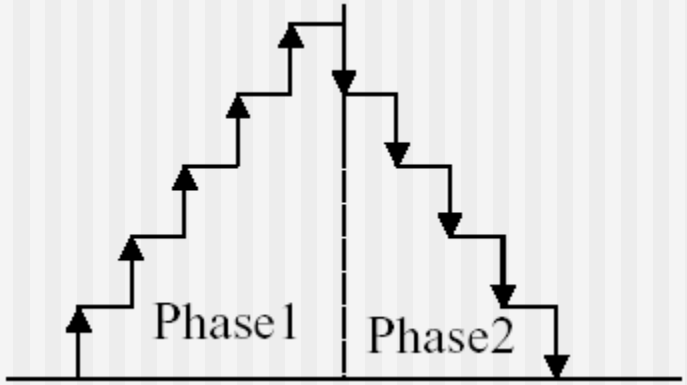
- Transactions indicate their intentions by requesting locks from the scheduler (lock manager).
- Every transaction that needs to access a data object for reading or writing must first lock the object.
- A transaction holds a lock until it explicitly releases it.
- Locks are either shared or exclusive.
- Shared and exclusive locks conflict

	Shared	Exclusive
Shared	Yes	No
Exclusive	No	No

- Locks allow concurrent processing of transactions.

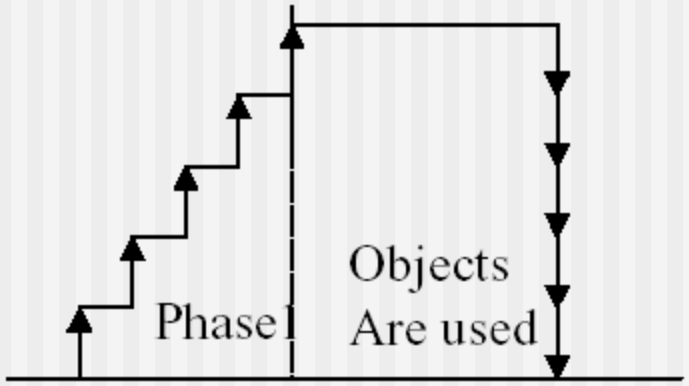
Two-Phase Locking Protocol

- *2PL*:
  - A transaction follows the 2PL protocol if all locking operations precede the first unlock operation in the transaction.
  - Phase 1 is the "*growing phase*" during which all the locks are requested
  - Phase 2 is the "*shrinking phase*" during which all locks are released



Strict Two-Phase Locking Protocol

- *Strict 2PL*:
  - All locks held by a transaction are released when the transaction completes (just before committing)
  - Strict 2PL allows only serializable schedules



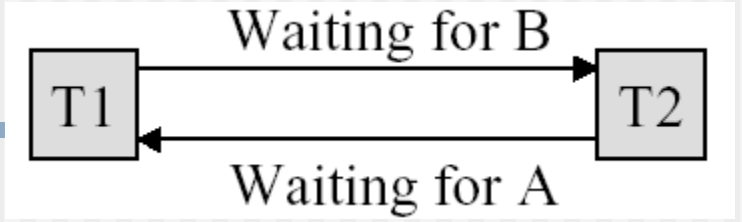
Lock Management

- Lock and unlock requests are handled by the lock manager
- Lock table entry:
  - Number of transactions currently holding a lock
  - Type of lock held (shared or exclusive)
  - Pointer to queue of lock requests
- Locking and unlocking have to be atomic operations
- Lock upgrade: transaction that holds a shared lock can be upgraded to hold an exclusive lock (also downgrade)

Deadlocks

- Deadlock: Cycle of transactions waiting for locks to be released by each other.
- A transaction is deadlocked if it is blocked and will remain blocked until intervention.
- Locking-based Concurrency Control algorithms may cause deadlocks.
- Two ways of dealing with deadlocks:
  - Deadlock prevention (guaranteeing no deadlocks or detecting deadlocks in advance before they occur)
  - Deadlock detection (allowing deadlocks to form and breaking them when they occur)

Deadlock Example



T1	T2
begin-transaction	
Write-lock(A)	begin-transaction
Read(A)	**Write-lock(B)**
A=A-100	Read(B)
Write(A)	B=B*1.06
Write-lock(B)	Write(B)
Wait	**write-lock(A)**
Wait	Wait
…	Wait
	…

Deadlock Prevention

- Assign priorities based on timestamps. (i.e. the oldest transaction has higher priority)
- Assume T_i wants a lock that T_j holds. Two policies are possible:
  - *Wait-Die*: If T_i has higher priority, T_i waits for T_j; otherwise T_i aborts
  - *Wound-wait*: If T_i has higher priority, T_j aborts; otherwise T_i waits
- If a transaction re-starts, make sure it has its original timestamp

Deadlock and Timeouts

- A simple approach to deadlock prevention (and *pseudo* detection) is based on lock timeouts
- After requesting a lock on a locked data object, a transaction waits, but if the lock is not granted within a period (timeout), a deadlock is assumed and the waiting transaction is aborted and re-started.
- Very simple practical solution adopted by many DBMSs.

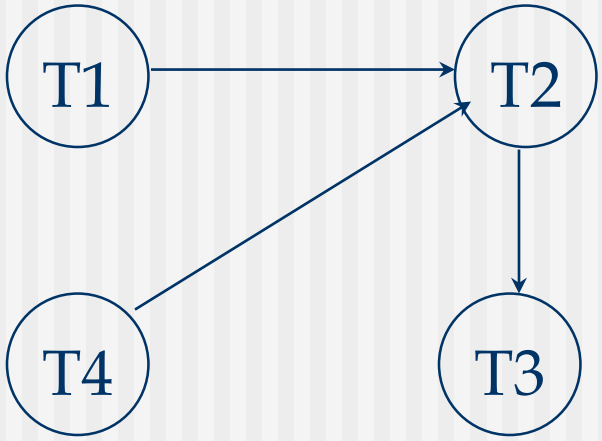
Deadlock Detection

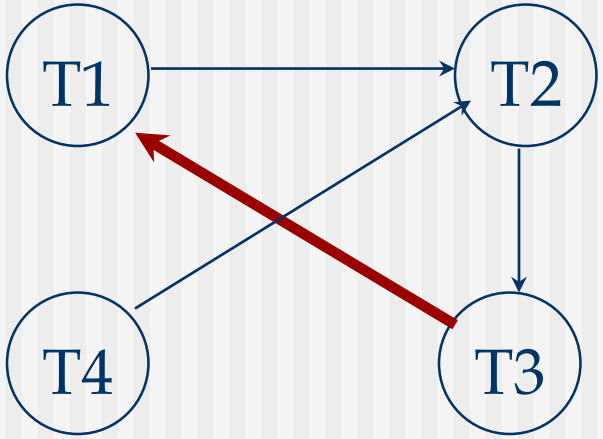
- Create a waits-for graph:
  - Nodes are transactions
  - There is an edge from T_i to T_j if T_i is waiting for T_j to release a lock
- Deadlock exists if there is a cycle in the graph.
- Periodically check for cycles in the waits-for graph

Deadlock Detection (cont.)

Example:

T1: S(A), R(A), S(B)
T2: X(B),W(B) X(C)
T3: S(C), R(C) X(A)
T4: X(B)





Recovery from Deadlock

- How to choose a a deadlock victim to abort?
  - How long the transaction has been running?
  - How many data objects have been updated?
  - How many data objects the transaction is still to update?
- Do we need to rollback the whole aborted transaction?
- Avoid starvation (when the same transaction is always the victim)