Database System
(SW5)

10. Transaction and Concurrency Control

Tiantian Liu
Department of Computer Science
Aalborg University
Fall 2025

Motivation

- Transaction boundaries are an important part of system design
- Users think in transactions
- Simple to handle multiple users

Learning Goals

- Understanding the transaction concept
- Understanding serializability
- Understand and use lock-based concurrency control
- Understand and use two-phase locking

Agenda

- Transaction
  - Transaction concept
  - Schedules
  - Conflict serializability
- Concurrency Control
  - Lock-based synchronization
  - Two-phase locking (2PL)
  - Deadlock

Transaction Concept

- Transaction to transfer 50 from account A to account B:
  - 1. read(A)
  - 2. A := A – 50
  - 3. write(A)
  - 4. read(B)
  - 5. B := B + 50
  - 6. write(B)
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

ACID Properties

- A **transaction** is a unit of program execution that accesses and possibly updates various data items.
- To preserve the integrity of data the database system must ensure:
  - **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
  - **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
  - **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions.
  - **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Operations on Transactions

- Begin of Transaction
  - Represents the beginning of a transaction, i.e., all following statements together form a transaction.
  - In SQL **BEGIN**;
- Commit
  - Represents the end of a transaction, i.e., all changes are made persistent and visible to others.
  - In SQL **COMMIT**;
- Rollback or Abort
  - Causes a transaction to roll back, i.e., all changes are undone/discarded.
  - In SQL **ROLLBACK**;

Savepoints

Long running transactions can specify savepoints.

- **SAVEPOINT** savepoint name;
- Defines a point/state within a transaction
- A transaction can be rolled back partially back up to the savepoint.

- **ROLLBACK TO** <savepoint name>:
- rolls the active transaction back to the savepoint <savepoint name>

Example

```
BEGIN;
INSERT INTO tab VALUES. . .
SAVEPOINT A;
INSERT INTO tab VALUES. . .
SAVEPOINT B;
SELECT * FROM tab;
ROLLBACK TO A;
SELECT * FROM tab;
. . .
```

Termination of Transactions

There are three possibilities for a transaction to terminate:

- Successful termination by **commit**
- Unsuccessful termination by **abort**
- Unsuccessful termination by a **failure**

How do DBMSs support transactions

The two most important components of transaction management are

- Multi-user synchronization (isolation)
  - Concurrency allows for high throughput
  - Serializability
- Recovery (atomicity and durability)
  - Roll back partially executed transactions
  - Re-executing transactions after failures
  - Guaranteeing persistence of transactional updates

Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system.
- Advantages are:
  - Increased processor and disk utilization, leading to better transaction throughput
    - E.g., one transaction can be using the CPU while another is reading from or writing to the disk
  - Reduced average response time for transactions: short transactions need not wait behind long ones.

Concurrent Executions

- Affects the "I" in ACID.
- The execution of multiple transactions T_1, T_2, and T_3
- (a) in a single-user environment

- (b) in a (concurrent) multi-user environment

Potential Problems

- Lost updates (overwriting updates)

Step	T_1	T_2
1	read(A, a_1)	
2	$a_1 := a_1 - 300$	
3		read(A, a_2)
4		$a_2 := a_2 * 1.03$
5		**write(A, a_2)**
6	**write(A, a_1)**	
7	read(B, b_1)	
8	$b_1 := b_1 + 300$	
9	write(B, b_1)	

Potential Problems

- Dirty read (dependency on non-committed updates)

Step	T_1	T_2
1	read(A, a_1)	
2	$a_1 := a_1 - 300$	
3	**write(A, a_1)**	
4		read(A, a_2)
5		$a_2 := a_2 * 1.03$
6		**write(A, a_2)**
7	read(B, b_1)	
8	…	
9	abort	

Potential Problems

- Non-repeatable read (dependency on other updates)

T_1	T_2
	select sum (balance)
	from account
update account	
set balance = 42000	
where accountID = 123	
	select sum (balance)
	from account

Potential Problems

- Phantom problem (dependency on new/deleted tuples)

T_1	T_2
	select sum (balance)
	from account
Insert into account	
values (C, 1000, …)	
	select sum (balance)
	from account

Agenda

- Transaction
  - Transaction concept
  - Schedules
  - Conflict serializability
- Concurrency Control
  - Lock-based synchronization
  - Two-phase locking (2PL)
  - Deadlock

Schedules

- Schedule – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
- A transaction that successfully completes its execution will have a **commit** instruction as the last statement
- A transaction that fails to successfully complete its execution will have an **abort** instruction as the last statement

Schedules

AALBORG UNIVERSITY

- Serial schedule
  - The operations of the transactions are executed sequentially with no overlap in time.
- Concurrent schedule
  - The operations of the transactions are executed with overlap in time.
- Valid schedule
  - A schedule is valid if the result of its execution is "correct".

Schedule 1

- Let T_1 transfer 50 from A to B, and T_2 transfer 10% of the balance from A to B.
- A serial schedule in which T_1 is followed by T_2:

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit

Schedule 2

- A serial schedule where T_2 is followed by T_1

T_1	T_2
	read (A) $temp := A * 0.1$ $A := A - temp$ write (A) read (B) $B := B + temp$ write (B) commit
read (A) $A := A - 50$ write (A) read (B) $B := B + 50$ write (B) commit	

Schedule 3

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is equivalent to Schedule 1

T_1	T_2
read (A) $A := A - 50$ write (A)	
	read (A) $temp := A * 0.1$ $A := A - temp$ write (A)
read (B) $B := B + 50$ write (B) commit	
	read (B) $B := B + temp$ write (B) commit

- In Schedules 1, 2 and 3, the sum A + B is preserved.

Schedule 4

- The following concurrent schedule does not preserve the value of (A + B).

T_1	T_2
read (A)	
$A := A - 50$	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	$B := B + temp$
	write (B)
	commit

Notion of Correctness

- Definition D1: A concurrent execution of transactions must leave the database in a consistent state.
- Definition D2: Concurrent execution of transactions must be (result) equivalent to some serial execution of the transactions.

Agenda

- Transaction
  - Transaction concept
  - Schedules
  - Conflict serializability
- Concurrency Control
  - Lock-based synchronization
  - Two-phase locking (2PL)
  - Deadlock

Serializability

- Basic Assumption – Each transaction preserves database consistency.
  - Thus, serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule.
- Different forms of schedule equivalence give rise to the notions of:
  - 1. Conflict serializability
  - 2. View serializability

Simplified View of Transactions

- We ignore operations other than **read** and **write** instructions
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.

Conflicting Instructions

- Instructions l_1 and l_2 of transactions T_1 and T_2 respectively, conflict if and only if there exists some item Q accessed by both l_1 and l_2, and at least one of these instructions wrote Q.
    1. l_1 = **read**(Q), l_2 = **read**(Q). l_1 and l_2 don't conflict.
    2. l_1 = **read**(Q), l_2 = **write**(Q). They conflict.
    3. l_1 = **write**(Q), l_2 = **read**(Q). They conflict
    4. l_1 = **write**(Q), l_2 = **write**(Q). They conflict
- If l_1 and l_2 are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflicts--Example

- Conflicts between pairs of transactions (T_1 and T_2) and their instructions.

T_1	T_2
read(X, x)	
	write(X, x)

Conflict

T_1	T_2
write(X, x)	
	read(X, x)

Conflict

T_1	T_2
write(X, x)	
	write(X, x)

Conflict

T_1	T_2
read(X, x)	
	read(X, x)

No conflict

Conflicts--Example

- Conflicts between pairs of transactions (T_1 and T_2) and their instructions.

T_1	T_2
read(X, x)	
	write(Y, y)

No conflict

T_1	T_2
write(X, x)	
	read(Y, y)

No conflict

T_1	T_2
write(X, x)	
	write(Y, y)

No conflict

T_1	T_2
read(X, x)	
	read(Y, y)

No conflict

Conflict Serializability

- If a schedule S_1 can be transformed into a schedule S_2 by a series of swaps of non-conflicting instructions, we say that S_1 and S_2 are conflict equivalent.
- We say that a schedule S is conflict serializable if it is conflict equivalent to a serial schedule

Conflict Serializability (Cont.)

AALBORG UNIVERSITY

- Let *I* and *J* be consecutive instructions of a schedule S_1 of multiple transactions.
- If *I* and *J* do not conflict, we can swap their order to produce a new schedule S_2.
- The instructions appear in the same order in S_1 and S_2, except for *I* and *J*, whose order does not matter.
- S_1 and S_2 are termed conflict equivalent schedules.

Conflict Serializability--Example

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Schedule 3

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Schedule 6

Conflict Serializability-Example

- Example of a schedule that is not conflict serializable:

T_3	T_4
read (Q)	
	write (Q)
write (Q)	

- We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

Conflict Serializability

- As the transformation shows, the initial concurrent schedule is conflict equivalent to a serial schedule and is therefore conflict serializable.

T_1:	R(A),		W(A),		R(B),	W(B),	Commit		
T_2:		R(C),		W(C),			R(A),	W(A),	Commit

T_1:	R(A),	W(A),			R(B),	W(B),	Commit		
T_2:			R(C),	W(C),			R(A),	W(A),	Commit

T_1:	R(A),	W(A),	R(B),	W(B),	Commit					
T_2:						R(C),	W(C),	R(A),	W(A),	Commit

Exercise 1

- The following schedules are conflict serializable or not:

schedule S_A	
T_1	T_2
read(Y, y)	
	read(X, x)
	write(X, x)
write(Y, y)	

schedule S_B	
T_3	T_4
read(X, x)	
	read(X, x)
	write(X, x)
write(X, x)	

schedule S_C	
T_5	T_6
read(X, x)	
	read(X, x)
write(X, x)	

schedule S_D	
T_7	T_8
read(X, x)	
	write(X, x)
write(X, x)	

Exercise 1--Solution

- The following schedules are conflict serializable or not:

schedule S_A	
T_1	T_2
read(Y, y)	
	read(X, x)
	write(X, x)
write(Y, y)	

conflict serializable

schedule S_B	
T_3	T_4
read(X, x)	
	read(X, x)
	write(X, x)
write(X, x)	

Not conflict serializable

schedule S_C	
T_5	T_6
read(X, x)	
	read(X, x)
write(X, x)	

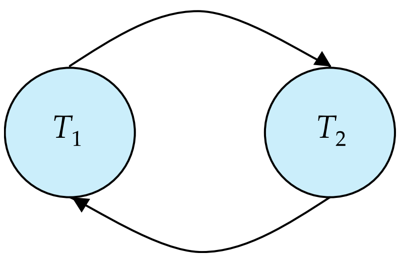
conflict serializable

schedule S_D	
T_7	T_8
read(X, x)	
	write(X, x)
write(X, x)	

Not conflict serializable

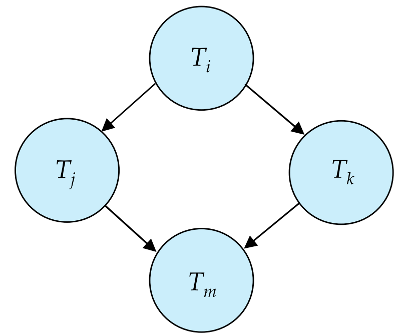
Conflict/Precedence Graph

- We construct a directed graph (conflict/precedence graph) for a schedule involving a set of transactions.
- Given a schedule for a set of transactions $T_1, T_2, \ldots, T_n$
  - The vertices of the conflict graph are the transaction identifiers.
  - An edge from T_i to T_j denotes that the two transactions are conflicting, with T_i making the relevant access earlier.
  - Sometimes the edge is labelled with the item involved in the conflict.

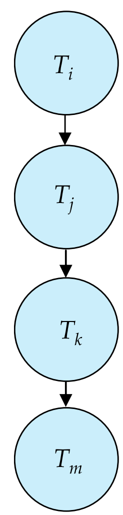


Test for Conflict Serializability

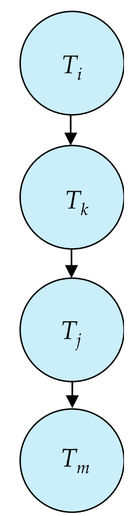
- A schedule is conflict serializable if and only if its precedence graph is **acyclic**.
- Intuitively, a conflict between two transactions forces an execution order between them (topological sorting)



(a)



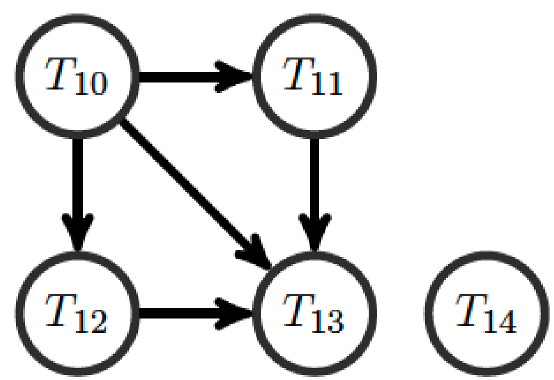
(b)



(c)

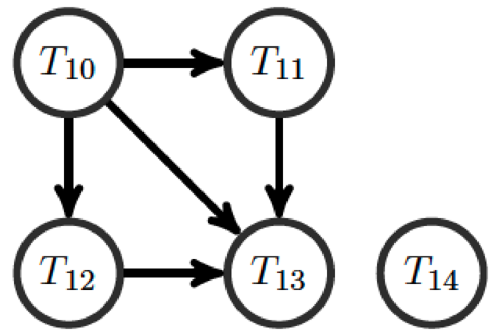
Conflict Graph--Example

T_{10}	T_{11}	T_{12}	T_{13}	T_{14}
	read(X, x)			
read(Y, y)				
read(Z, z)				
				read(V, v)
				read(W, w)
				write(W, w)
	read(Y, y)			
	write(Y, y)			
		read(Z, z)		
		write(Z, z)		
read(T, t)				
			read(Y, y)	
			write(Y, y)	
			read(Z, z)	
			write(Z, z)	
read(U, u)				



Conflict Graph--Example

T_{10}	T_{11}	T_{12}	T_{13}	T_{14}
	read(X, x)			
read(Y, y)				
read(Z, z)				
				read(V, v)
				read(W, w)
				write(W, w)
	read(Y, y)			
	write(Y, y)			
		read(Z, z)		
		write(Z, z)		
read(T, t)				
			read(Y, y)	
			write(Y, y)	
			read(Z, z)	
			write(Z, z)	
read(U, u)				

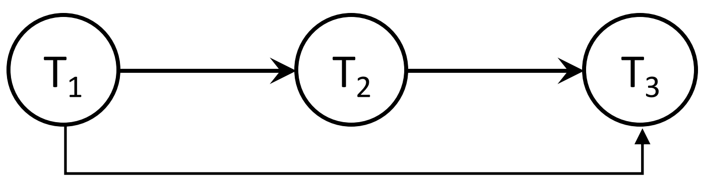


- Which of the following are conflict equivalent serial schedules?
- T_{10}, T_{11}, T_{12}, T_{13}, and T_{14} Yes
- T_{14}, T_{10}, T_{12}, T_{11}, and T_{13} Yes
- T_{14}, T_{13}, T_{12}, T_{11}, and T_{10} No

Example

- Draw the precedence graphs of the following schedule:
  - Decide if the schedule is conflict serializable. If it is conflict serializable, give one example of a conflict equivalent serial schedule?

T_1:	R(A), W(A),			R(B), W(B)		
T_2:		W(A),			R(B), W(B)	
T_3:			R(A),			W(A)



Conflict equivalent serial schedule: T_1, T_2, T_3

Exercise 2

- Draw the precedence graphs of the following schedules:
  - Decide if the schedule is conflict serializable. If it is conflict serializable, give one example of a conflict equivalent serial schedule?

T_1:	R(A), W(A),			R(B), W(B)	
T_2:		R(C), W(A)			
T_3:			R(B),		W(C)

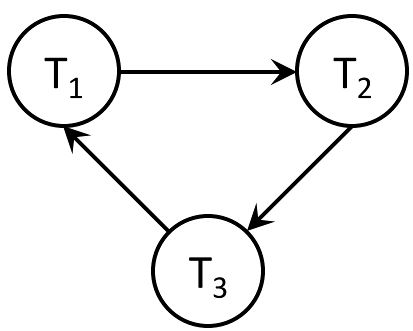
T_4:	R(A),		W(A)	
T_5:		W(A)		
T_6:				W(A)

Exercise 2--Solution

AALBORG UNIVERSITY

- Draw the precedence graphs of the following schedules:
  - Decide if the schedule is conflict serializable. If it is conflict serializable, give one example of a conflict equivalent serial schedule?

T_1:	R(A), W(A),			R(B), W(B)	
T_2:		R(C), W(A)			
T_3:			R(B),		W(C)

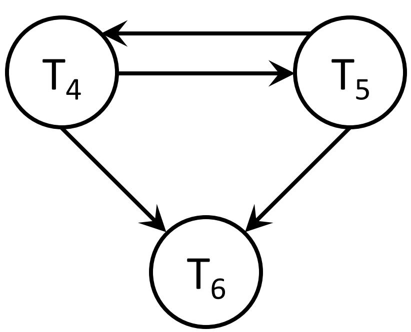


Not Conflict Serializable

Exercise 2--Solution

- Draw the precedence graphs of the following schedules:
  - Decide if the schedule is conflict serializable. If it is conflict serializable, give one example of a conflict equivalent serial schedule?

T_4:	R(A),		W(A)	
T_5:		W(A)		
T_6:				W(A)



Not Conflict Serializable

Agenda

- Transaction
  - Transaction concept
  - Schedules
  - Conflict serializability
- Concurrency Control
  - Lock-based synchronization
  - Two-phase locking (2PL)
  - Deadlock

Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item.
- Data items can be locked in two modes :
  - 1. **Exclusive** (X) mode. X-lock is requested using **lock-X** instruction.
  - 2. **Shared** (S) mode. S-lock is requested using **lock-S** instruction.
- Operations on locks
  - **lock-S(Q)**: set shared lock on data item Q. Allow multiple transactions to read Q simultaneously.
  - **lock-X(Q)**: set exclusive lock on data item Q. Ensure only one transaction can read or write Q at a time.
  - **unlock(Q)**: release lock on data item Q.

Lock-Based Protocols (Cont.)

- Lock-compatibility matrix

	S	X
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item
- But if any transaction holds an exclusive on the item no other transaction may hold any lock on the item

Example

- Example of a transaction performing locking:

  T_2: **lock-S**(*A*);
  read (*A*);
  unlock(*A*);

  lock-S(*B*);
  read (*B*);
  unlock(*B*);
  display(*A+B*)

- Locking as above is not sufficient to guarantee serializability

Example

- T_1 transfers 50 kr. from account B to account A.
- T_2 displays the total amount of money in accounts A and B.
- Initially A = 100 and B = 200

Result of the serial execution:
T_2 will display 300
Using this schedule:
T_2 will display 250

T_1	T_2
lock-X(B)	
read(B, b)	
b ← b - 50	lock-S(A)
write(B, b)	read(A, a)
unlock(B)	unlock(A)
lock-X(A)	lock-S(B)
read(A, a)	read(B, b)
a ← a + 50	unlock(B)
write(A, a)	display(a + b)
display(a + b)	
unlock(A)	

Problems with early unlocking

- T_1 transfers 50 kr. from account B to account A.
- T_2 displays the total amount of money in accounts A and B.
- Initially A = 100 and B = 200

T_1	T_2
lock-X(B)	
read(B, b)	
b ← b - 50	lock-S(A)
write(B, b)	read(A, a)
unlock(B)	unlock(A)
lock-X(A)	lock-S(B)
read(A, a)	read(B, b)
a ← a + 50	unlock(B)
write(A, a)	display(a + b)
unlock(A)	

Early unlocking can cause incorrect results (non-serializable schedules) but allows for a higher degree of concurrency.

Problems with late unlocking

- T_1 transfers 50 kr. from account B to account A.
- T_2 displays the total amount of money in accounts A and B.
- Initially A = 100 and B = 200

Late unlocking avoids non-serializable schedules.
But it increases the chances of deadlocks.

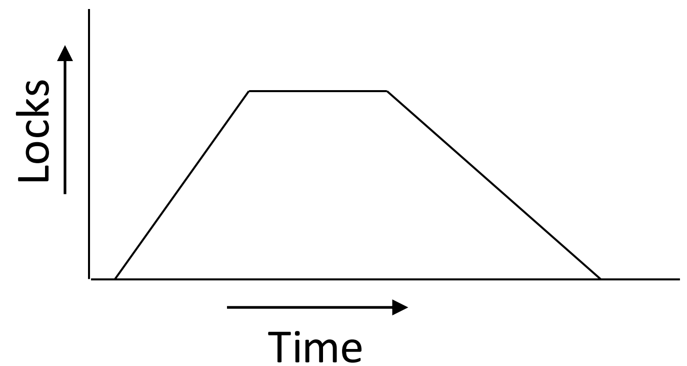
T_1	T_2
lock-X(B)	
read(B, b)	
b ← b - 50	**lock-S(A)**
write(B, b)	read(A, a)
~~unlock(B)~~	~~unlock(A)~~
lock-X(A)	**lock-S(B)**
read(A, a)	read(B, b)
a ← a + 50	~~unlock(B)~~
write(A, a)	display(a + b)
unlock(B)	**unlock(A)**
unlock(A)	**unlock(B)**

Agenda

- Transaction
  - Transaction concept
  - Schedules
  - Conflict serializability
- Concurrency Control
  - Lock-based synchronization
  - Two-phase locking (2PL)
  - Deadlock

The Two-Phase Locking Protocol

- A protocol which ensures conflict-serializable schedules.
- Phase 1: Growing Phase
  - Transaction may obtain locks
  - Transaction may not release locks
- Phase 2: Shrinking Phase
  - Transaction may release locks
  - Transaction may not obtain locks



2PL: yes or no?

T_i
lock-X(A)
lock-X(B)
lock-X(C)
unlock(A)
unlock(B)
unlock(C)

yes

T_i
lock-X(A)
lock-X(B)
unlock(B)
lock-X(C)
unlock(A)
unlock(C)

no

Agenda

- Transaction
  - Transaction concept
  - Schedules
  - Conflict serializability
- Concurrency Control
  - Lock-based synchronization
  - Two-phase locking (2PL)
  - Deadlock

Deadlock Handling

- System is **deadlocked** if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

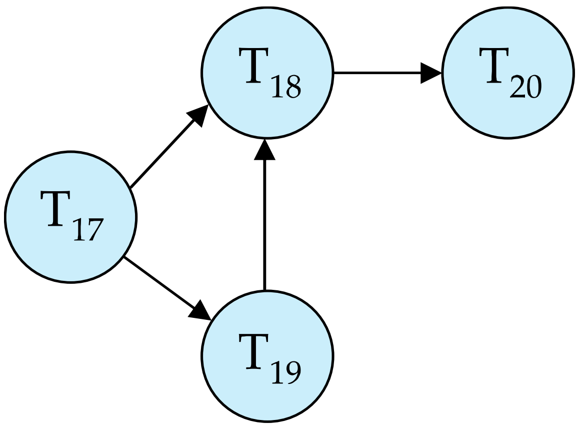
Deadlock Handling

- Deadlock prevention protocols ensure that the system will never enter into a deadlock state. Some prevention strategies:
  - Pre-declaration
  - Graph-based protocol

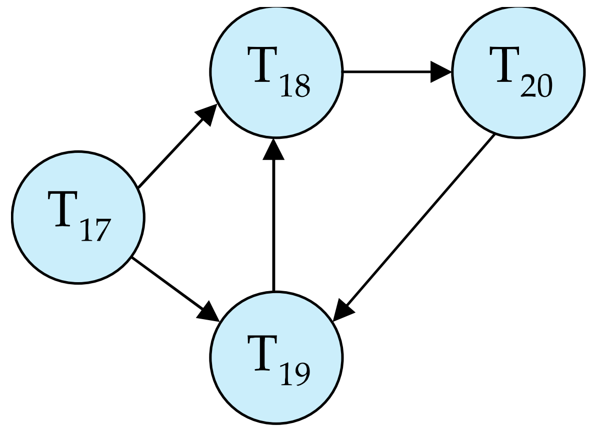
Deadlock Detection

- Wait-for graph
  - Vertices: transactions
  - Edge from $T_i \rightarrow T_j$: if T_i is waiting for a lock held in conflicting mode by T_j
- The system is in a deadlock state if and only if the wait-for graph has a cycle.
- Invoke a deadlock-detection algorithm periodically to look for cycles.

Wait-for Graph



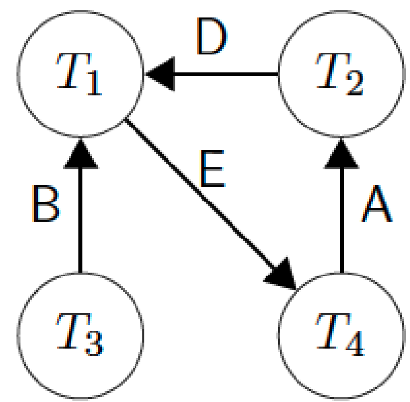
Wait-for graph without a cycle



Wait-for graph with a cycle

Wait-for Graph Example

T_1	T_2	T_3	T_4
	lock-X(A)		
lock-X(B)			
lock-X(C)			
		lock-X(B)	
lock-X(D)			
			lock-X(E)
	lock-X(D)		
			lock-X(A)
lock-X(E)			



- Rollback of one or multiple involved transactions to release the deadlock

Exercise 3

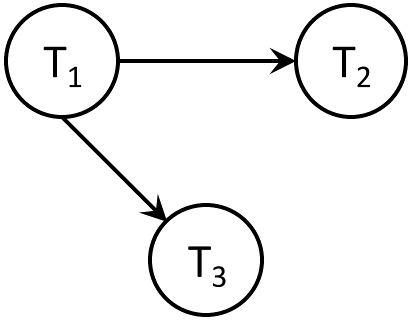
- Draw the wait-for graphs of the following schedules:
  - Decide if there is a deadlock

T_1	T_2	T_3
	lock-X(A)	
		lock-X(B)
		lock-X(C)
lock-X(A)		
lock-X(B)		
		lock-X(D)

Exercise 3--Solution

- Draw the wait-for graphs of the following schedules:
  - Decide if there is a deadlock

T_1	T_2	T_3
	lock-X(A)	
		lock-X(B)
		lock-X(C)
lock-X(A)		
lock-X(B)		
		lock-X(D)



Deadlock Recovery

- When deadlock is detected :
  - Some transaction will have to rolled back (made a victim) to break deadlock cycle.
  - Rollback -- determine how far to roll back transaction
    - Total rollback: Abort the transaction and then restart it.
    - Partial rollback: Roll back victim transaction only as far as necessary to release locks that another transaction in cycle is waiting for

Summary

- Transaction
  - Transaction concept
  - Schedules
  - Conflict serializability
- Concurrency Control
  - Lock-based synchronization
  - Two-phase locking (2PL)
  - Deadlock

Next Lecture

- Lecture Session
  - Recovery
    - Understanding basic logging algorithms
    - Understanding the importance of atomicity and durability
  - Summary
- Exercise Session
  - Mock exam
  - Q & A