



# Database Management Systems

## Module 31: Transactions/1

**Partha Pratim Das**

*Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur*

[ppd@cse.iitkgp.ernet.in](mailto:ppd@cse.iitkgp.ernet.in)

**Srijoni Majumdar  
Himadri B G S Bhuyan  
Gurunath Reddy M**



**Database System Concepts, 6<sup>th</sup> Ed.**

©Silberschatz, Korth and Sudarshan  
[www.db-book.com](http://www.db-book.com)



# Week 06 Recap

- **Module 26: Indexing and Hashing/1 (Indexing/1)**
  - Basic Concepts of Indexing
  - Ordered Indices
- **Module 27: Indexing and Hashing/2 (Indexing/2)**
  - Balanced Binary Search Trees
  - 2-3-4 Tree
- **Module 28: Indexing and Hashing/3 (Indexing/3)**
  - B+-Tree Index Files
  - B-Tree Index Files
- **Module 29: Indexing and Hashing/4 (Hashing)**
  - Static Hashing
  - Dynamic Hashing
  - Comparison of Ordered Indexing and Hashing
  - Bitmap Indices
- **Module 30: Indexing and Hashing/5 (Index Design)**
  - Index Definition in SQL
  - Guidelines for Indexing

NPTEL



# Module Objectives

- To understand the concept of transaction – ‘doing a task in a database’ and its state
- To explore issues in concurrent execution of transactions

NPTEL



# Module Outline

- Transaction Concept
- Transaction State
- Concurrent Executions

NPTEL



- **Transaction Concept**
- Transaction State
- Concurrent Executions

NPTEL

# TRANSACTION CONCEPT



# Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items
- For example, 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



# Required Properties of a Transaction

## ■ Atomicity requirement

- If the transaction fails after step 3 and before step 6, money will be “lost” leading to an inconsistent database state
  - ▶ Failure could be due to software or hardware
- The system should ensure that updates of a partially executed transaction are not reflected in the database

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)**



# Required Properties of a Transaction

## ■ Consistency requirement

- In example, the sum of A and B is unchanged by the execution of the transaction
- In general, consistency requirements include
  - ▶ Explicitly specified integrity constraints
    - primary keys and foreign keys
  - ▶ Implicit integrity constraints
    - sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
- A transaction, when starting to execute, must see a consistent database
- During transaction execution the database may be temporarily inconsistent
- When the transaction completes successfully the database must be consistent
  - ▶ Erroneous transaction logic can lead to inconsistency

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)**



# Required Properties of a Transaction (Cont.)

## ■ Isolation requirement

- If between steps 3 and 6 (of the fund transfer transaction) , another transaction **T2** is allowed to access the partially updated database, it will see an inconsistent database (the sum  $A + B$  will be less than it should be).

**T1**

1. **read(A)**
2.  $A := A - 50$
3. **write(A)**

**T2**

read(A), read(B), print(A+B)

4. **read(B)**
5.  $B := B + 50$
6. **write(B)**

- Isolation can be ensured trivially by running transactions **serially**
  - ▶ That is, one after the other
- However, executing multiple transactions concurrently has significant benefits



# Required Properties of a Transaction

## ■ Durability requirement

- Once the user has been notified that the transaction has completed (i.e., the transfer of the \$50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures

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)**



# 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. Intermediate transaction results must be hidden from other concurrently executed transactions
  - That is, for every pair of transactions  $T_i$  and  $T_j$ , it appears to  $T_i$  that either  $T_j$  finished execution before  $T_i$  started, or  $T_j$  started execution after  $T_i$  finished

- **Durability:**

- After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures



- Transaction Concept
- **Transaction State**
- Concurrent Executions

NPTEL

# TRANSACTION STATE



# Transaction State

- **Active**

- The initial state; the transaction stays in this state while it is executing

- **Partially committed**

- After the final statement has been executed

- **Failed**

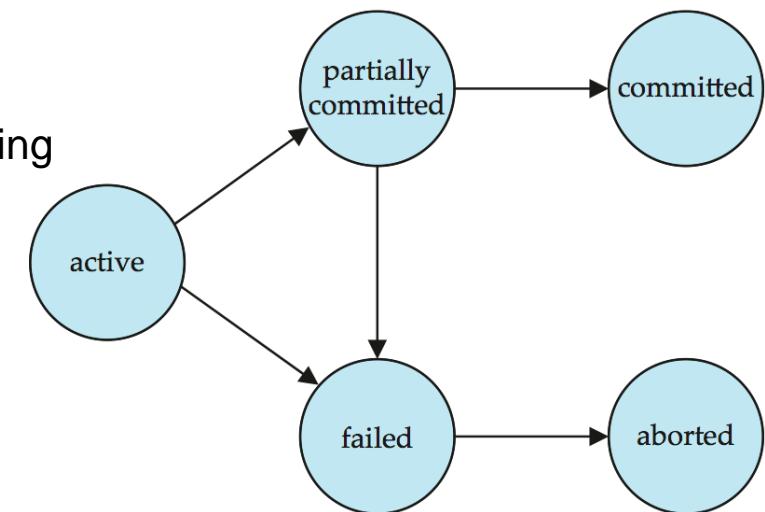
- After the discovery that normal execution can no longer proceed

- **Aborted**

- After the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
    - ▶ Restart the transaction
      - can be done only if no internal logical error
    - ▶ Kill the transaction

- **Committed**

- After successful completion

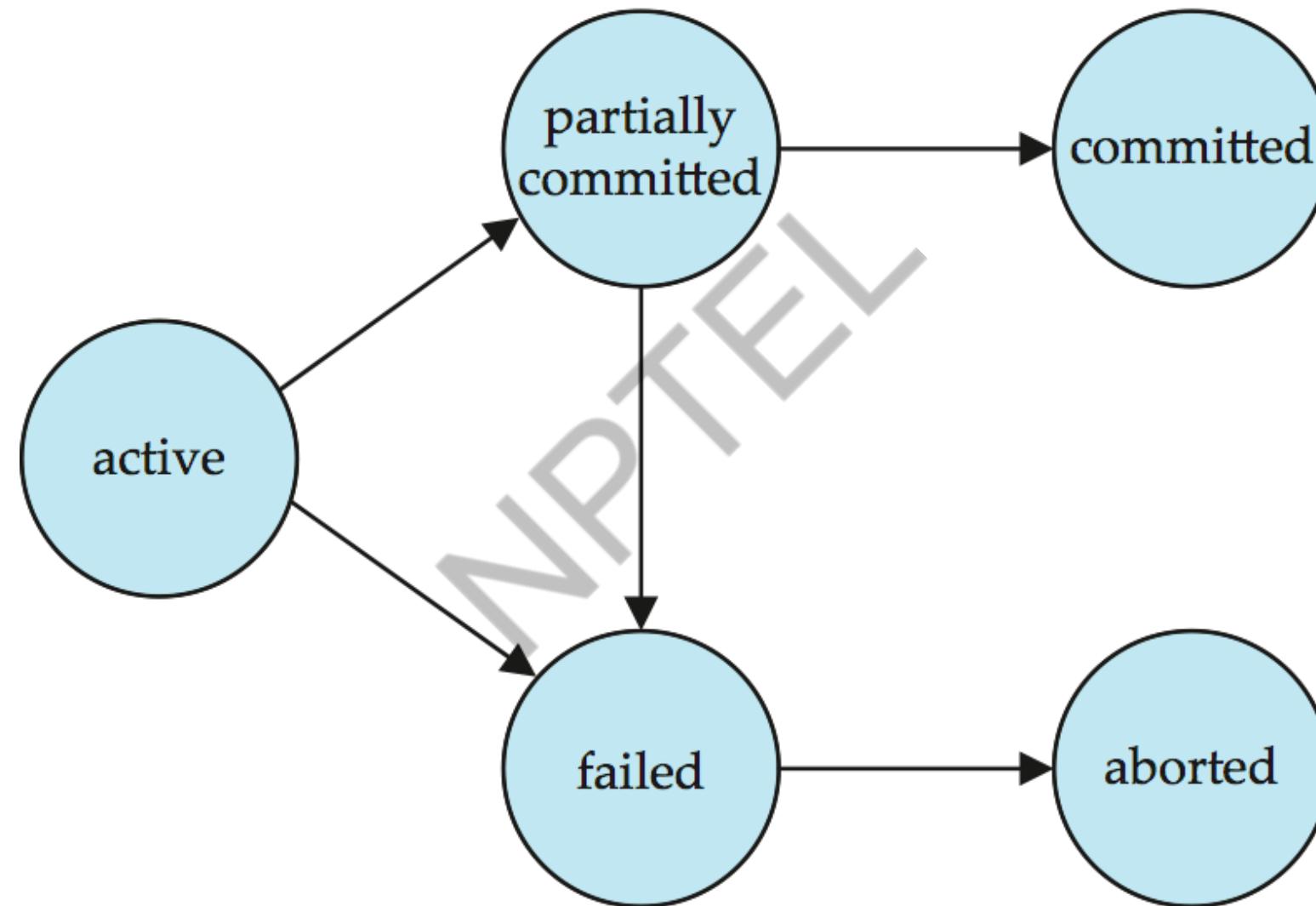


Transaction to transfer \$50 from account A to account B:

1. **read(A)**
2.  $A := A - 50$
3. **write(A)**
4. **read(B)**
5.  $B := B + 50$
6. **write(B)**



# Transitions for Transaction State





- Transaction Concept
- Transaction State
- **Concurrent Executions**

NPTEL

# CONCURRENT EXECUTIONS



# Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **Increased processor and disk utilization**, leading to better transaction *throughput*
    - ▶ For example, 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
- **Concurrency control schemes** – mechanisms to achieve isolation
  - That is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database



# Schedules

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - A schedule for a set of transactions must consist of all instructions of those transactions
  - Must preserve the order in which the instructions appear in each individual transaction
- A transaction that successfully completes its execution will have a **commit** instructions as the last statement
  - By default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an **abort** instruction as the last statement



# Schedule 1

- Let  $T_1$  transfer \$50 from  $A$  to  $B$ , and  $T_2$  transfer 10% of the balance from  $A$  to  $B$
- An example of 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

A	B	A+B	Transaction	Remarks
100	200	300	@ Start	
50	200	250	T1, write A	
50	250	300	T1, write B	@ Commit
45	250	295	T2, write A	
45	255	300	T2, write B	@ Commit

Consistent @ Commit  
Inconsistent @ Transit  
Inconsistent @ Commit



# Schedule 2

- A **serial** schedule in which  $T_2$  is followed by  $T_1$  :

$T_1$	$T_2$
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

A	B	A+B	Transaction	Remarks
100	200	300	@ Start	
90	200	290	T2, write A	
90	210	300	T2, write B	@ Commit
40	210	250	T1, write A	
40	260	300	T1, write B	@ Commit

Consistent @ Commit  
Inconsistent @ Transit  
Inconsistent @ Commit

Values of A & B are different from  
Schedule 1 – yet consistent



# 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

$T_1$	$T_2$	$T_1$	$T_2$	A	B	$A+B$	Transaction	Remarks
read ( $A$ ) $A := A - 50$ write ( $A$ )	read ( $A$ ) $temp := A * 0.1$ $A := A - temp$ write ( $A$ )	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	100	200	300	@ Start	
				50	200	250	$T_1$ , write A	
				45	200	245	$T_2$ , write A	
				45	250	295	$T_1$ , write B	@ Commit
				45	255	300	$T_2$ , write B	@ Commit
<b>Schedule 3</b>				<b>Consistent @ Commit</b> <b>Inconsistent @ Transit</b> <b>Inconsistent @ Commit</b>				
<b>Schedule 1</b>								

Note – In schedules 1, 2 and 3, the sum “ $A + B$ ” is preserved



# Schedule 4

- The following concurrent schedule does not preserve the sum 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

A	B	A+B	Transaction	Remarks
100	200	300	@ Start	
90	200	290	T2, write A	
90	200	290	T1, write A	
90	250	340	T1, write B	@ Commit
90	260	350	T2, write B	@ Commit

Consistent @ Commit  
Inconsistent @ Transit  
Inconsistent @ Commit



# Module Summary

- A task is a database is done as a transaction that passes through several states
- Transactions are executed in concurrent fashion for better throughput
- Concurrent execution of transactions raise serializability issues that need to be addressed
- All schedules may not satisfy ACID properties

NPTEL



# Instructor and TAs

Name	Mail	Mobile
Partha Pratim Das, Instructor	ppd@cse.iitkgp.ernet.in	9830030880
Srijoni Majumdar, TA	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, TA	himadribhuyan@gmail.com	9438911655
Gurunath Reddy M	mgurunathreddy@gmail.com	9434137638

**Slides used in this presentation are borrowed from <http://db-book.com/> with kind permission of the authors.**

**Edited and new slides are marked with “PPD”.**



# Database Management Systems

## Module 32: Transactions/2: Serializability

**Partha Pratim Das**

*Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur*

[ppd@cse.iitkgp.ernet.in](mailto:ppd@cse.iitkgp.ernet.in)

**Srijoni Majumdar  
Himadri B G S Bhuyan  
Gurunath Reddy M**



**Database System Concepts, 6<sup>th</sup> Ed.**

©Silberschatz, Korth and Sudarshan  
[www.db-book.com](http://www.db-book.com)



# Module Recap

- Transaction Concept
- Transaction State
- Concurrent Executions

NPTEL



# Module Objectives

- To understand the issues that arise when two or more transactions work concurrently
- To introduce the notions of Serializability that ensure schedules for transactions that may run in concurrent fashion but still guarantee and serial behavior
- To analyze the conditions, called conflicts, that need to be honored to attain Serializable schedules

NPTEL



# Module Outline

- Serializability
- Conflict Serializability

NPTEL



- **Serializability**
- Conflict Serializability

# SERIALIZABILITY

NPTEL



# 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**

NPTEL



# Simplified view of transactions

- We ignore operations other than **read** and **write** instructions
  - Other operations happen in memory (are temporary in nature) and (mostly) do not affect the state of the database
  - This is a simplifying assumption for analysis
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes
- Our simplified schedules consist of only **read** and **write** instructions



# Conflicting Instructions

- Let  $I_i$  and  $I_j$  be two Instructions of transactions  $T_i$  and  $T_j$  respectively. Instructions  $I_i$  and  $I_j$  **conflict** if and only if there exists some item  $Q$  accessed by both  $I_i$  and  $I_j$ , and at least one of these instructions wrote  $Q$ 
  1.  $I_i = \text{read}(Q)$ ,  $I_j = \text{read}(Q)$ .  $I_i$  and  $I_j$  don't conflict
  2.  $I_i = \text{read}(Q)$ ,  $I_j = \text{write}(Q)$ . They conflict
  3.  $I_i = \text{write}(Q)$ ,  $I_j = \text{read}(Q)$ . They conflict
  4.  $I_i = \text{write}(Q)$ ,  $I_j = \text{write}(Q)$ . They conflict
- Intuitively, a conflict between  $I_i$  and  $I_j$  forces a (logical) temporal order between them
  - If  $I_i$  and  $I_j$  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



- Serializability
- Conflict  
Serializability

NPTEL

# CONFLICT SERIALIZABILITY



# Conflict Serializability

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

NPTEL



# Conflict Serializability (Cont.)

- Schedule 3 can be transformed into Schedule 6 – a serial schedule where  $T_2$  follows  $T_1$ , by a series of swaps of non-conflicting instructions.
  - Swap  $T_1.\text{read}(B)$  and  $T_2.\text{write}(A)$
  - Swap  $T_1.\text{read}(B)$  and  $T_2.\text{read}(A)$
  - Swap  $T_1.\text{write}(B)$  and  $T_2.\text{write}(A)$
  - Swap  $T_1.\text{write}(B)$  and  $T_2.\text{read}(A)$
- Therefore, Schedule 3 is conflict serializable:

$T_1$	$T_2$
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 5

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

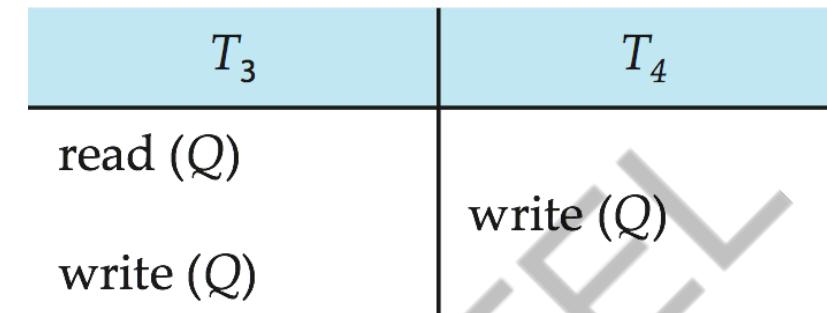
Schedule 6

*These swaps do not conflict as they work with different items (A or B) in different transactions.*



# Conflict Serializability (Cont.)

- Example of a schedule that is not conflict serializable:



- We are unable to swap instructions in the above schedule to obtain either the serial schedule  $\langle T_3, T_4 \rangle$ , or the serial schedule  $\langle T_4, T_3 \rangle$



# Example: Bad Schedule

- Consider two transactions:

## Transaction 1

**UPDATE** accounts

**SET** balance = balance - 100

**WHERE** acct\_id = 31414

- In terms of read / write we can write these as:

Transaction 1:  $r_1(A)$ ,  $w_1(A)$  // A is the balance for acct\_id = 31414

Transaction 2:  $r_2(A)$ ,  $w_2(A)$ ,  $r_2(B)$ ,  $w_2(B)$  // B is balance of other accounts

- Consider schedule S:

- Schedule S:  $r_1(A)$ ,  $r_2(A)$ ,  $w_1(A)$ ,  $w_2(A)$ ,  $r_2(B)$ ,  $w_2(B)$
- Suppose: A starts with \$200, and account B starts with \$100

- Schedule S is very bad! (At least, it's bad if you're the bank!) We withdrew \$100 from account A, but somehow the database has recorded that our account now holds \$201!

## Transaction 2

**UPDATE** accounts

**SET** balance = balance \* 1.005

A      B

(initial:) 200.00 100.00

$r_1(A)$ :

$r_2(A)$ :

$w_1(A)$ : 100.00

$w_2(A)$ : 201.00

$r_2(B)$ :

$w_2(B)$ : 100.50

**Schedule S**

Source: <http://www.cburch.com/cs/340/reading/serial/>



# Example: Bad Schedule

- Ideal schedule is serial:

Serial schedule 1:  $r_1(A), w_1(A), r_2(A), w_2(A), r_2(B), w_2(B)$

Serial schedule 2:  $r_2(A), w_2(A), r_2(B), w_2(B), r_1(A), w_1(A)$

- We call a schedule **serializable** if it has the same effect as some serial schedule regardless of the specific information in the database.
- As an example, consider Schedule  $T$ , which has swapped the third and fourth operations from  $S$ :
  - Schedule  $S$ :  $r_1(A), r_2(A), \textcolor{red}{w_1(A)}, \textcolor{red}{w_2(A)}, r_2(B), w_2(B)$
  - Schedule  $T$ :  $r_1(A), r_2(A), \textcolor{blue}{w_2(A)}, \textcolor{blue}{w_1(A)}, r_2(B), w_2(B)$
- By first example, the outcome is the same as Serial schedule 1. But that's just a peculiarity of the data, as revealed by the second example, where the final value of A can't be the consequence of either of the possible serial schedules.
- So neither  $S$  nor  $T$  are serializable

	Schedule 1		Schedule 2	
	A	B	A	B
Initial Values	200.00	100.00	200.00	100.00
Final Values	100.50	100.50	101.00	100.50
Initial Values	100.00	100.00	100.00	100.00
Final Values	0.00	100.50	1.00	100.50

$A$  is \$100 initially

$A$        $B$

(initial:) 100.00 100.00

$r_1(A)$ :

$r_2(A)$ :

$w_2(A)$ : 100.50

$w_1(A)$ : 0.00

$r_2(B)$ :

$w_2(B)$ :

$A$  is \$200 initially

$A$        $B$

(initial:) 200.00 100.00

$r_1(A)$ :

$r_2(A)$ :

$w_2(A)$ : 201.00

$w_1(A)$ : 100.00

$r_2(B)$ :

$w_2(B)$ :

100.50

**Schedule T**

Source: <http://www.cburch.com/cs/340/reading/serial/>



# Example: Good Schedule

- What's a non-serial example of a serializable schedule?
  - We could credit interest to A first, then withdraw the money, then credit interest to B:
  - Schedule  $U$ :  $r_2(A)$ ,  $w_2(A)$ ,  $r_1(A)$ ,  $w_1(A)$ ,  $r_2(B)$ ,  $w_2(B)$ 
    - ▶ Initial: A = 200, B = 100
    - ▶ Final: A = 101, B = 100.50
- Schedule U is conflict serializable to Schedule 2:

Schedule  $U$ :  $r_2(A)$ ,  $w_2(A)$ ,  $r_1(A)$ ,  $w_1(A)$ ,  $r_2(B)$ ,  $w_2(B)$

swap  $w_1(A)$  and  $r_2(B)$ :  $r_2(A)$ ,  $w_2(A)$ ,  $r_1(A)$ ,  $r_2(B)$ ,  $w_1(A)$ ,  $w_2(B)$

swap  $w_1(A)$  and  $w_2(B)$ :  $r_2(A)$ ,  $w_2(A)$ ,  $r_1(A)$ ,  $r_2(B)$ ,  $w_2(B)$ ,  $w_1(A)$

swap  $r_1(A)$  and  $r_2(B)$ :  $r_2(A)$ ,  $w_2(A)$ ,  $r_2(B)$ ,  $r_1(A)$ ,  $w_2(B)$ ,  $w_1(A)$

swap  $r_1(A)$  and  $w_2(B)$ :  $r_2(A)$ ,  $w_2(A)$ ,  $r_2(B)$ ,  $w_2(B)$ ,  $r_1(A)$ ,  $w_1(A)$ : Schedule 2

Source: <http://www.cburch.com/cs/340/reading/serial/>



# Serializability

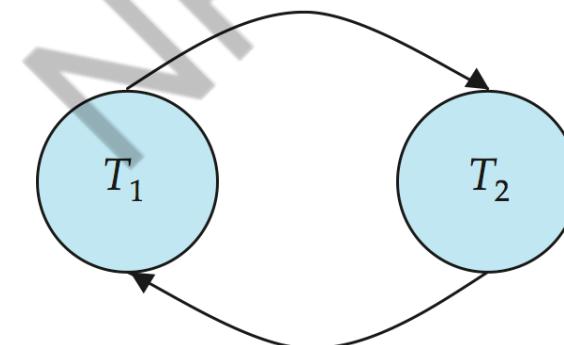
- Are all serializable schedules conflict-serializable? No.
- Consider the following schedule for a set of three transactions.
  - $w_1(A), w_2(A), w_2(B), w_1(B), w_3(B)$
- We can perform no swaps to this:
  - The first two operations are both on  $A$  and at least one is a write;
  - The second and third operations are by the same transaction;
  - The third and fourth are both on  $B$  at least one is a write; and
  - So are the fourth and fifth.
  - So this schedule is not conflict-equivalent to anything – and certainly not any serial schedules.
- However, since nobody ever reads the values written by the  $w_1(A)$ ,  $w_2(B)$ , and  $w_1(B)$  operations, the schedule has the same outcome as the serial schedule:
  - $w_1(A), w_1(B), w_2(A), w_2(B), w_3(B)$

Source: <http://www.cburch.com/cs/340/reading/serial/>



# Precedence Graph

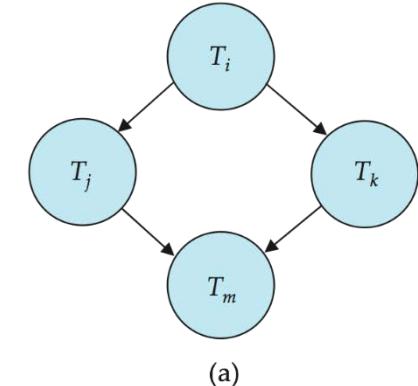
- Consider some schedule of a set of transactions  $T_1, T_2, \dots, T_n$
- **Precedence graph**
  - A direct graph where the vertices are the transactions (names)
- We draw an arc from  $T_i$  to  $T_j$  if the two transactions conflict, and  $T_i$  accessed the data item on which the conflict arose earlier
- We may label the arc by the item that was accessed
- **Example**



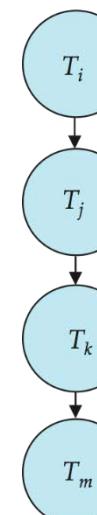


# Testing for Conflict Serializability

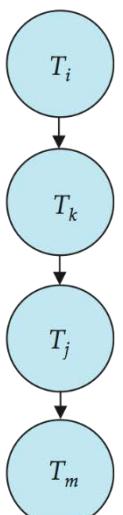
- A schedule is conflict serializable if and only if its precedence graph is acyclic
- Cycle-detection algorithms exist which take order  $n^2$  time, where  $n$  is the number of vertices in the graph
  - (Better algorithms take order  $n + e$  where  $e$  is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph
  - That is, a linear order consistent with the partial order of the graph.
  - For example, a serializability order for the schedule (a) would be one of either (b) or (c)



(a)



(b)



(c)



# Testing for Conflict Serializability

- Build a directed graph, with a vertex for each transaction.
- Go through each operation of the schedule.
  - If the operation is of the form  $w_i(X)$ , find each subsequent operation in the schedule also operating on the same data element  $X$  by a different transaction: that is, anything of the form  $r_j(X)$  or  $w_j(X)$ . For each such subsequent operation, add a directed edge in the graph from  $T_i$  to  $T_j$ .
  - If the operation is of the form  $r_i(X)$ , find each subsequent *write* to the same data element  $X$  by a different transaction: that is, anything of the form  $w_j(X)$ . For each such subsequent write, add a directed edge in the graph from  $T_i$  to  $T_j$ .
- The schedule is conflict-serializable if and only if the resulting directed graph is acyclic.
- Moreover, we can perform a topological sort on the graph to discover the serial schedule to which the schedule is conflict-equivalent.



# Testing for Conflict Serializability

- Consider the following schedule:
- We start with an empty graph with five vertices labeled  $T_1, T_2, T_3, T_4, T_5$ .
- We go through each operation in the schedule:

$w_1(A)$ :  $A$  is subsequently read by  $T_2$ , so add edge  $T_1 \rightarrow T_2$

$r_2(A)$ : no subsequent writes to  $A$ , so no new edges

$w_1(B)$ :  $B$  is subsequently read by  $T_4$ , so add edge  $T_1 \rightarrow T_4$

$w_3(C)$ :  $C$  is subsequently read by  $T_2$ , so add edge  $T_3 \rightarrow T_2$

$r_2(C)$ : no subsequent writes to  $C$ , so no new edges

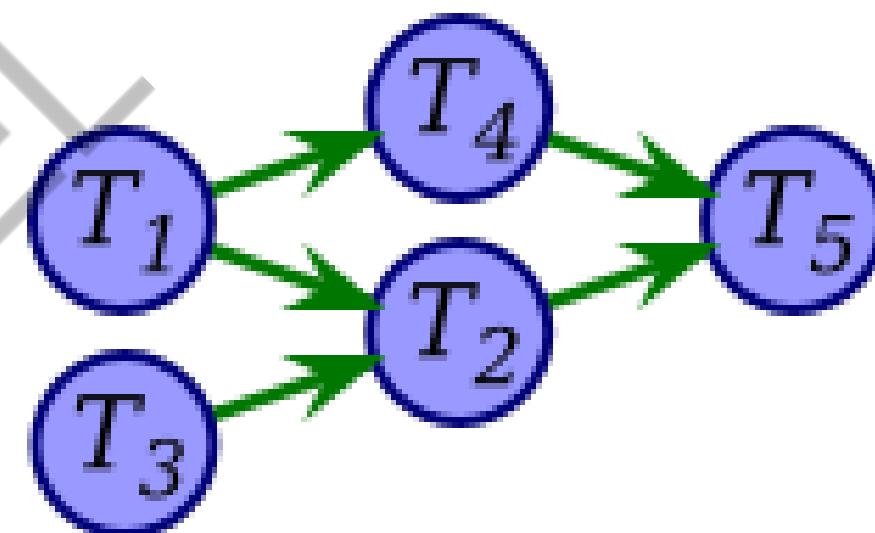
$r_4(B)$ : no subsequent writes to  $B$ , so no new edges

$w_2(D)$ :  $C$  is subsequently read by  $T_2$ , so add edge  $T_3 \rightarrow T_2$

$w_4(E)$ :  $E$  is subsequently written by  $T_5$ , so add edge  $T_4 \rightarrow T_5$

$r_5(D)$ : no subsequent writes to  $D$ , so no new edges

$w_5(E)$ : no subsequent operations on  $E$ , so no new edges



- We end up with precedence graph
- This graph has no cycles, so the original schedule must be serializable. Moreover, since one way to topologically sort the graph is  $T_3-T_1-T_4-T_2-T_5$ , one serial schedule that is conflict-equivalent is
  - $w_3(C), w_1(A), w_1(B), r_4(B), w_4(E), r_2(A), r_2(C), w_2(D), r_5(D), w_5(E)$

Source: <http://www.cburch.com/cs/340/reading/serial/>



# Module Summary

- Understood the issues that arise when two or more transactions work concurrently
- Learnt the forms of serializability in terms of conflict and view serializability
- Acyclic precedence graph can ensure conflict serializability

NPTEL



# Instructor and TAs

Name	Mail	Mobile
Partha Pratim Das, Instructor	ppd@cse.iitkgp.ernet.in	9830030880
Srijoni Majumdar, TA	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, TA	himadribhuyan@gmail.com	9438911655
Gurunath Reddy M	mgurunathreddy@gmail.com	9434137638

**Slides used in this presentation are borrowed from <http://db-book.com/> with kind permission of the authors.**

**Edited and new slides are marked with “PPD”.**



# Database Management Systems

## Module 33: Transactions/3: Recoverability

**Partha Pratim Das**

*Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur*

[ppd@cse.iitkgp.ernet.in](mailto:ppd@cse.iitkgp.ernet.in)

**Srijoni Majumdar  
Himadri B G S Bhuyan  
Gurunath Reddy M**



**Database System Concepts, 6<sup>th</sup> Ed.**

©Silberschatz, Korth and Sudarshan  
[www.db-book.com](http://www.db-book.com)



# Module Recap

- Serializability
- Conflict Serializability

NPTEL



# Module Objectives

- What happens if system fails while a transaction is in execution? Can a consistent state be reached for the database? Recoverability attempts to answer issues in state and transaction recovery in the face of system failures
- Conflict serializability is a crisp concept for concurrent execution that guarantees ACID properties and has a simple detection algorithm. Yet only few schedules are Conflict serializable in practice. There is a need to explore – View Serializability – a weaker system for better concurrency



# Module Outline

- Recoverability and Isolation
- Transaction Definition in SQL
- View Serializability
- Complex Notions of Serializability

NPTEL



- **Recoverability and Isolation**
- Transaction Definition in SQL
- View Serializability
- Complex Notions of Serializability

NPTEL

# RECOVERABILITY AND ISOLATION



# What is recovery?

- Serializability helps to ensure Isolation and Consistency of a schedule
- Yet, the Atomicity and Consistency may be compromised in the face of system failures
- Consider a schedule comprising a single transaction (obviously serial):
  1. **read(A)**
  2.  $A := A - 50$
  3. **write(A)**
  4. **read(B)**
  5.  $B := B + 50$
  6. **write(B)**
  7. **commit // Make the changes permanent; show the results to the user**
- What if system fails after Step 3 and before Step 6?
  - Leads to inconsistent state
  - Need to rollback update of A
- This is known as **Recovery**

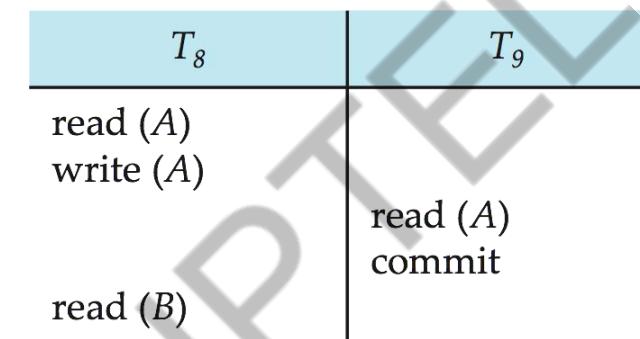


# Recoverable Schedules

## ■ Recoverable schedule

- If a transaction  $T_j$  reads a data item previously written by a transaction  $T_i$ , then the commit operation of  $T_i$  **must** appear before the commit operation of  $T_j$ .

- The following schedule is not recoverable if  $T_9$  commits immediately after the read(A) operation



- If  $T_8$  should abort,  $T_9$  would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable



# Cascading Rollbacks

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

$T_{10}$	$T_{11}$	$T_{12}$
read (A) read (B) write (A)		
abort	read (A) write (A)	read (A)

- If  $T_{10}$  fails,  $T_{11}$  and  $T_{12}$  must also be rolled back
- Can lead to the undoing of a significant amount of work



# Cascadeless Schedules

- **Cascadeless schedules** — for each pair of transactions  $T_i$  and  $T_j$  such that  $T_j$  reads a data item previously written by  $T_i$ , the commit operation of  $T_i$  appears before the read operation of  $T_j$ .
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless
- Example of a schedule that is NOT cascadeless

$T_{10}$	$T_{11}$	$T_{12}$
read (A) read (B) write (A)  abort	read (A) write (A)	read (A)



# Recoverable Schedules: Example

## ■ Irrecoverable Schedule

T1	T1's Buffer	T2	T2's Buffer	Database
				A = 5000
R(A);	A = 5000			A = 5000
A = A - 1000;	A = 4000			A = 5000
W(A);	A = 4000			A = 4000
		R(A);	A = 4000	A = 4000
		A = A + 500;	A = 4500	A = 4000
		W(A);	A = 4500	A = 4500
		Commit;		
Failure Point				
Commit;				

Source: <https://www.geeksforgeeks.org/dbms-recoverability-of-schedules/>



# Recoverable Schedules: Example

- Recoverable Schedule with cascading rollback

T1	T1's Buffer	T2	T2's Buffer	Database
				A = 5000
R(A);	A = 5000			A = 5000
A = A - 1000;	A = 4000			A = 5000
W(A);	A = 4000			A = 4000
		R(A);	A = 4000	A = 4000
		A = A + 500;	A = 4500	A = 4000
		W(A);	A = 4500	A = 4500
Failure Point				
Commit;				
		Commit;		

Source: <https://www.geeksforgeeks.org/dbms-recoverability-of-schedules/>



# Recoverable Schedules: Example

- Recoverable Schedule without cascading rollback

T1	T1's Buffer	T2	T2's Buffer	Database
				A = 5000
R(A);	A = 5000			A = 5000
A = A - 1000;	A = 4000			A = 5000
W(A);	A = 4000			A = 4000
Commit;				
		R(A);	A = 4000	A = 4000
		A = A + 500;	A = 4500	A = 4000
		W(A);	A = 4500	A = 4500
		Commit;		

Source: <https://www.geeksforgeeks.org/dbms-recoverability-of-schedules/>



- Recoverability and Isolation
- **Transaction Definition in SQL**
- View Serializability
- Complex Notions of Serializability

NPTEL

# TRANSACTION DEFINITION IN SQL



# Transaction Definition in SQL

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction
- In SQL, a transaction begins implicitly
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one
  - **Rollback work** causes current transaction to abort
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully
  - Implicit commit can be turned off by a database directive
    - ▶ For example in JDBC, `connection.setAutoCommit(false);`



# Transaction Control Language (TCL)

- The following commands are used to control transactions.
  - **COMMIT** – to save the changes
  - **ROLLBACK** – to roll back the changes
  - **SAVEPOINT** – creates points within the groups of transactions in which to ROLLBACK
  - **SET TRANSACTION** – Places a name on a transaction
- Transactional control commands are only used with the **DML Commands** such as
  - INSERT, UPDATE and DELETE only
  - They cannot be used while creating tables or dropping them because these operations are automatically committed in the database

Source: <http://www.tutorialspoint.com/sql/sql-transactions.htm>



# TCL: COMMIT Command

- The COMMIT is the transactional command used to save changes invoked by a transaction to the database
- The COMMIT saves all the transactions to the database since the last COMMIT or ROLLBACK command
- The syntax for the COMMIT command is as follows:
  - SQL> DELETE FROM Customers WHERE AGE = 25;
  - SQL> COMMIT;

SQL> SELECT \* FROM Customers;

**Before DELETE**

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000
2	Khilan	25	Delhi	1500
3	kaushik	23	Kota	2000
4	Chaitali	25	Mumbai	6500
5	Hardik	27	Bhopal	8500
6	Komal	22	MP	4500
7	Muffy	24	Indore	10000

**After DELETE**

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000
3	kaushik	23	Kota	2000
5	Hardik	27	Bhopal	8500
6	Komal	22	MP	4500
7	Muffy	24	Indore	10000

Source: <http://www.tutorialspoint.com/sql/sql-transactions.htm>



# TCL: ROLLBACK Command

- The ROLLBACK is the command used to undo transactions that have not already been saved to the database
- This can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued
- The syntax for a ROLLBACK command is as follows:
  - SQL> DELETE FROM Customers WHERE AGE = 25;
  - SQL> ROLLBACK;

SQL> SELECT \* FROM Customers;

**Before DELETE**

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000
2	Khilan	25	Delhi	1500
3	kaushik	23	Kota	2000
4	Chaitali	25	Mumbai	6500
5	Hardik	27	Bhopal	8500
6	Komal	22	MP	4500
7	Muffy	24	Indore	10000

**After DELETE**

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000
2	Khilan	25	Delhi	1500
3	kaushik	23	Kota	2000
4	Chaitali	25	Mumbai	6500
5	Hardik	27	Bhopal	8500
6	Komal	22	MP	4500
7	Muffy	24	Indore	10000

Source: <http://www.tutorialspoint.com/sql/sql-transactions.htm>



# TCL: SAVEPOINT / ROLLBACK Command

- A SAVEPOINT is a point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction
- The syntax for a SAVEPOINT command is:
  - `SAVEPOINT SAVEPOINT_NAME;`
- This command serves only in the creation of a SAVEPOINT among all the transactional statements.
- The ROLLBACK command is used to undo a group of transactions
- The syntax for rolling back to a SAVEPOINT is:
  - `ROLLBACK TO SAVEPOINT_NAME;`
- Example:
  - SQL> `SAVEPOINT SP1;`
    - ▶ Savepoint created.
  - SQL> `DELETE FROM Customers WHERE ID=1;`
    - ▶ 1 row deleted.
  - SQL> `SAVEPOINT SP2;`
    - ▶ Savepoint created.
  - SQL> `DELETE FROM Customers WHERE ID=2;`
    - ▶ 1 row deleted.
  - SQL> `SAVEPOINT SP3;`
    - ▶ Savepoint created.
  - SQL> `DELETE FROM Customers WHERE ID=3;`
    - ▶ 1 row deleted.

Source: <http://www.tutorialspoint.com/sql/sql-transactions.htm>



# TCL: SAVEPOINT / ROLLBACK Command

- Three records deleted
- Undo the deletion of first two
- SQL> ROLLBACK TO SP2;
  - Rollback complete

```
SQL> SAVEPOINT SP1;
SQL> DELETE FROM Customers WHERE ID=1;
SQL> SAVEPOINT SP2;
SQL> DELETE FROM Customers WHERE ID=2;
SQL> SAVEPOINT SP3;
SQL> DELETE FROM Customers WHERE ID=3;
```

**At the beginning**

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000
2	Khilan	25	Delhi	1500
3	kaushik	23	Kota	2000
4	Chaitali	25	Mumbai	6500
5	Hardik	27	Bhopal	8500
6	Komal	22	MP	4500
7	Muffy	24	Indore	10000

**After ROLLBACK**

ID	NAME	AGE	ADDRESS	SALARY
2	Khilan	25	Delhi	1500
3	kaushik	23	Kota	2000
4	Chaitali	25	Mumbai	6500
5	Hardik	27	Bhopal	8500
6	Komal	22	MP	4500
7	Muffy	24	Indore	10000

Source: <http://www.tutorialspoint.com/sql/sql-transactions.htm>



# TCL: RELEASE SAVEPOINT Command

- The RELEASE SAVEPOINT command is used to remove a SAVEPOINT that you have created
- The syntax for a RELEASE SAVEPOINT command is as follows.
  - RELEASE SAVEPOINT SAVEPOINT\_NAME;
- Once a SAVEPOINT has been released, you can no longer use the ROLLBACK command to undo transactions performed since the last SAVEPOINT

Source: <http://www.tutorialspoint.com/sql/sql-transactions.htm>



# TCL: SET TRANSACTION Command

- The SET TRANSACTION command can be used to initiate a database transaction
- This command is used to specify characteristics for the transaction that follows
  - For example, you can specify a transaction to be read only or read write
- The syntax for a SET TRANSACTION command is as follows:
  - SET TRANSACTION [ READ WRITE | READ ONLY ];

Source: <http://www.tutorialspoint.com/sql/sql-transactions.htm>



- Recoverability and Isolation
- Transaction Definition in SQL
- **View Serializability**
- Complex Notions of Serializability

NPTEL

# VIEW SERIALIZABILITY



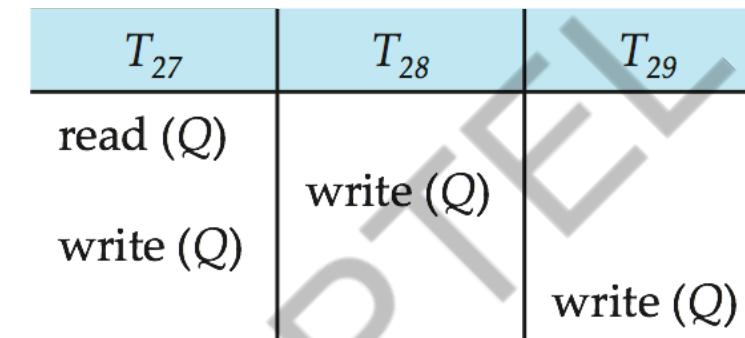
# View Serializability

- Let  $S$  and  $S'$  be two schedules with the same set of transactions.  $S$  and  $S'$  are **view equivalent** if the following three conditions are met, for each data item  $Q$ ,
  1. If in schedule  $S$ , transaction  $T_i$  reads the initial value of  $Q$ , then in schedule  $S'$  also transaction  $T_i$  must read the initial value of  $Q$ .
  2. If in schedule  $S$  transaction  $T_i$  executes **read**( $Q$ ), and that value was produced by transaction  $T_j$  (if any), then in schedule  $S'$  also transaction  $T_i$  must read the value of  $Q$  that was produced by the same **write**( $Q$ ) operation of transaction  $T_j$ .
  3. The transaction (if any) that performs the final **write**( $Q$ ) operation in schedule  $S$  must also perform the final **write**( $Q$ ) operation in schedule  $S'$
- As can be seen, view equivalence is also based purely on **reads** and **writes** alone



# View Serializability (Cont.)

- A schedule  $S$  is **view serializable** if it is view equivalent to a serial schedule
- Every conflict serializable schedule is also view serializable
- Below is a schedule which is view-serializable but *not* conflict serializable



- What serial schedule is above equivalent to?
  - $T_{27}-T_{28}-T_{29}$
  - The one read( $Q$ ) instruction reads the initial value of  $Q$  in both schedules and
  - $T_{29}$  performs the final write of  $Q$  in both schedules
- $T_{28}$  and  $T_{29}$  perform write( $Q$ ) operations called **blind writes**, without having performed a read( $Q$ ) operation
- Every view serializable schedule that is not conflict serializable has **blind writes**



# Test for View Serializability

- The precedence graph test for conflict serializability cannot be used directly to test for view serializability
  - Extension to test for view serializability has cost exponential in the size of the precedence graph
- The problem of checking if a schedule is view serializable falls in the class of *NP*-complete problems
  - Thus, existence of an efficient algorithm is *extremely* unlikely
- However, practical algorithms that just check some **sufficient conditions** for view serializability can still be used

NPTEL



# View Serializability: Example 1

- Check whether the schedule is view serializable or not?
  - S : R2(B); R2(A); R1(A); R3(A); W1(B); W2(B); W3(B);
- Solution:
  - With 3 transactions, total number of schedules possible =  $3! = 6$ 
    - <T1 T2 T3>
    - <T1 T3 T2>
    - <T2 T3 T1>
    - <T2 T1 T3>
    - <T3 T1 T2>
    - <T3 T2 T1>
  - Final update on data items :
    - A : -
    - B : T1 T2 T3
    - Since the final update on B is made by T3, so the transaction T3 must execute after transactions T1 and T2.
    - Therefore,  $(T1, T2) \rightarrow T3$ . Now, Removing those schedules in which T3 is not executing at last:
      - <T1 T2 T3>
      - <T2 T1 T3>

NPTEL



# View Serializability: Example 1

- Check whether the schedule is view serializable or not?
  - S : R2(B); R2(A); R1(A); R3(A); W1(B); W2(B); W3(B);
- Solution:
  - Initial Read + Which transaction updates after read?
    - A : T2 T1 T3 (initial read)
    - B : T2 (initial read); T1 (update after read)
    - The transaction T2 reads B initially which is updated by T1. So T2 must execute before T1.
    - Hence,  $T2 \rightarrow T1$ . Removing those schedules in which T2 is executing before T1:
    - $\langle T2 \ T1 \ T3 \rangle$
  - Write Read Sequence (WR)
    - No need to check here
  - Hence, view equivalent serial schedule is:
    - **$T2 \rightarrow T1 \rightarrow T3$**



# View Serializability: Example 2

- Check whether the schedule is Conflict serializable and view serializable or not?
  - S : R1(A); R2(A); R3(A); R4(A); W1(B); W2(B); W3(B); W4(B)
- Solution is given in the next slide (hidden). First try to solve is and then check the solution.

NPTEL



- Recoverability and Isolation
- Transaction Definition in SQL
- View Serializability
- **Complex Notions of Serializability**

# COMPLEX NOTIONS OF SERIALIZABILITY



# More Complex Notions of Serializability

- The schedule below produces the same outcome as the serial schedule  $\langle T_1, T_5 \rangle$ , yet is not conflict equivalent or view equivalent to it

$T_1$	$T_5$
read ( $A$ ) $A := A - 50$ write ( $A$ )	
read ( $B$ ) $B := B + 50$ write ( $B$ )	read ( $B$ ) $B := B - 10$ write ( $B$ )

- If we start with  $A = 1000$  and  $B = 2000$ , the final result is 960 and 2040
- Determining such equivalence requires analysis of operations other than read and write



# Module Summary

- With proper planning, a database can be recovered back to a consistent state from inconsistent state in the face of system failures. Such a recovery is done via cascaded or cascadeless rollback
- View Serializability is a weaker serializability system for better concurrency. However, testing for view serializability is NP complete

NPTEL



# Instructor and TAs

Name	Mail	Mobile
Partha Pratim Das, Instructor	ppd@cse.iitkgp.ernet.in	9830030880
Srijoni Majumdar, TA	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, TA	himadribhuyan@gmail.com	9438911655
Gurunath Reddy M	mgurunathreddy@gmail.com	9434137638

**Slides used in this presentation are borrowed from <http://db-book.com/> with kind permission of the authors.**

**Edited and new slides are marked with “PPD”.**



# Database Management Systems

## Module 34: Concurrency Control/1

**Partha Pratim Das**

*Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur*

[ppd@cse.iitkgp.ernet.in](mailto:ppd@cse.iitkgp.ernet.in)

**Srijoni Majumdar  
Himadri B G S Bhuyan  
Gurunath Reddy M**



**Database System Concepts, 6<sup>th</sup> Ed.**

©Silberschatz, Korth and Sudarshan  
[www.db-book.com](http://www.db-book.com)



# Module Recap

- Recoverability and Isolation
- Transaction Definition in SQL
- View Serializability
- Complex Notions of Serializability

NPTEL



# Module Objectives

- Concurrency Control through design of serializable schedule is difficult in general. Hence we take a look into locking mechanism and Lock-Based Protocols
- We need to understand how locks may be implemented

NPTEL



# Module Outline

- Concurrency Control
- Lock-Based Protocols
- Implementing Locking

NPTEL



- **Concurrency Control**
- Lock-Based Protocols
- Implementing Locking

NPTEL

# CONCURRENCY CONTROL



# Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are both:
  - Conflict serializable
  - Recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur
- Testing a schedule for serializability *after* it has executed is a little too late!
  - Tests for serializability help us understand why a concurrency control protocol is correct
- **Goal** – to develop concurrency control protocols that will assure serializability



# Concurrency Control

- One way to ensure isolation is to require that data items be accessed in a mutually exclusive manner; that is, while one transaction is accessing a data item, no other transaction can modify that data item
  - Should a transaction hold a lock on the whole database
    - ▶ Would lead to strictly serial schedules – very poor performance
- The most common method used to implement locking requirement is to allow a transaction to access a data item only if it is currently holding a **lock** on that item



- Concurrency Control
- **Lock-Based Protocols**
- Implementing Locking

# LOCK-BASED PROTOCOLS



# 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*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction
  2. *shared (S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction
- A transaction can unlock a data item Q by the **unlock(Q)** Instruction
- Lock requests are made to the concurrency-control manager by the programmer
- Transaction can proceed only after request is granted



# Lock-Based Protocols

## ■ 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
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted
- Transaction  $T_i$  may unlock a data item that it had locked at some earlier point
- Note that a transaction must hold a lock on a data item as long as it accesses that item
- Moreover, it is not necessarily desirable for a transaction to unlock a data item immediately after its final access of that data item, since serializability may not be ensured



# Lock-Based Protocols: Example

- Let  $A$  and  $B$  be two accounts that are accessed by transactions  $T1$  and  $T2$ .
  - Transaction  $T1$  transfers \$50 from account  $B$  to account  $A$ .
  - Transaction  $T2$  displays the total amount of money in accounts  $A$  and  $B$ , that is, the sum  $A + B$
  - Suppose that the values of accounts  $A$  and  $B$  are \$100 and \$200, respectively

$T1:$

```
lock-X( $B$ );  
read( $B$ );  
 $B := B - 50$ ;  
write( $B$ );  
unlock( $B$ );  
lock-X( $A$ );  
read( $A$ );  
 $A := A + 50$ ;  
write( $A$ );  
unlock( $A$ );
```

$T2:$

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

- If these transactions are executed serially, either as  $T1$ ,  $T2$  or the order  $T2$ ,  $T1$ , then transaction  $T2$  will display the value \$300



# Lock-Based Protocols: Example

- If, however, these transactions are executed concurrently, then schedule 1 is possible
- In this case, transaction  $T_2$  displays \$250, which is incorrect. The reason for this mistake is that
  - the transaction  $T_1$  unlocked data item  $B$  too early, as a result of which  $T_2$  saw an inconsistent state
- Suppose we delay unlocking till the end

$T_1$ :

```
lock-X(B);  
read(B);  
 $B := B - 50$ ;  
write(B);  
unlock(B);  
lock-X(A);  
read(A);  
 $A := A + 50$ ;  
write(A);  
unlock(A);
```

$T_2$ :

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

$T_1$	$T_2$	concurrency-control manager
lock-X(B)		grant-X(B, $T_1$ )
read(B) $B := B - 50$ write(B) unlock(B)	lock-s(A) read(A) unlock(A) lock-s(B)	grant-s(A, $T_2$ )
	read(B) unlock(B) display(A + B)	grant-s(B, $T_2$ )
	lock-X(A) read(A) $A := A - 50$ write(A) unlock(A)	grant-X(A, $T_1$ )

Schedule 1



# Lock-Based Protocols: Example

- Delaying unlocking till the end, T1 becomes T3 and T2 becomes T4

T3:

```
lock-X(B);
read(B);
B := B - 50;
write(B);
lock-X(A);
read(A);
A := A + 50;
write(A);
unlock(B);
unlock(A)
```

T4:

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

- Hence, sequence of reads and writes as in Schedule 1 is no longer possible
- T4 will correctly display \$300

$T_1$	$T_2$	concurrency-control manager
lock-X(B)		grant-x(B, $T_1$ )
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
lock-S(A)		grant-s(A, $T_2$ )
read(A)		
unlock(A)		
lock-S(B)		
read(B)		grant-s(B, $T_2$ )
unlock(B)		
display(A + B)		
lock-X(A)		grant-x(A, $T_1$ )
read(A)		
$A := A - 50$		
write(A)		
unlock(A)		

Schedule 1



# Lock-Based Protocols: Example

- Given, T3 and T4, consider Schedule 2 (partial)
- Since  $T_3$  is holding an exclusive mode lock on  $B$  and  $T_4$  is requesting a shared-mode lock on  $B$ ,  $T_4$  is waiting for  $T_3$  to unlock  $B$
- Similarly, since  $T_4$  is holding a shared-mode lock on  $A$  and  $T_3$  is requesting an exclusive-mode lock on  $A$ ,  $T_3$  is waiting for  $T_4$  to unlock  $A$
- Thus, we have arrived at a state where neither of these transactions can ever proceed with its normal execution
- This situation is called **deadlock**
- When deadlock occurs, the system must roll back one of the two transactions.
- Once a transaction has been rolled back, the data items that were locked by that transaction are unlocked
- These data items are then available to the other transaction, which can continue with its execution

$T_3$ :

```
lock-X( $B$ );  
read( $B$ );  
 $B := B - 50$ ;  
write( $B$ );  
lock-X( $A$ );  
read( $A$ );  
 $A := A + 50$ ;  
write( $A$ );  
unlock( $B$ );  
unlock( $A$ )
```

$T_4$ :

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

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

Schedule 2



# Lock-Based Protocols

- If we do not use locking, or if we unlock data items too soon after reading or writing them, we may get inconsistent states
- On the other hand, if we do not unlock a data item before requesting a lock on another data item, deadlocks may occur
- Deadlocks are a necessary evil associated with locking, if we want to avoid inconsistent states
- Deadlocks are definitely preferable to inconsistent states, since they can be handled by rolling back transactions, whereas inconsistent states may lead to real-world problems that cannot be handled by the database system
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks
- Locking protocols restrict the set of possible schedules
- The set of all such schedules is a proper subset of all possible serializable schedules
- We present locking protocols that allow only conflict-serializable schedules, and thereby ensure isolation



# The Two-Phase Locking Protocol

- This protocol 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
- The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points**
  - That is, the point where a transaction acquired its final lock



# The Two-Phase Locking Protocol (Cont.)

- There can be conflict serializable schedules that cannot be obtained if two-phase locking is used
- However, in the absence of extra information (e.g., ordering of access to data), two-phase locking is needed for conflict serializability in the following sense:
  - Given a transaction  $T_i$  that does not follow two-phase locking, we can find a transaction  $T_j$  that uses two-phase locking, and a schedule for  $T_i$  and  $T_j$  that is not conflict serializable

NPTEL



# Lock Conversions

- Two-phase locking with lock conversions:
  - First Phase:
    - can acquire a lock-S on item
    - can acquire a lock-X on item
    - can convert a lock-S to a lock-X (upgrade)
  - Second Phase:
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (downgrade)
- This protocol assures serializability. But still relies on the programmer to insert the various locking instructions



# Automatic Acquisition of Locks: Read

- A transaction  $T_i$  issues the standard read/write instruction, without explicit locking calls
- The operation **read( $D$ )** is processed as:

**if**  $T_i$  has a lock on  $D$

**then**

read( $D$ )

**else begin**

if necessary wait until no other transaction has a **lock-X** on  $D$

grant  $T_i$  a **lock-S** on  $D$ ;

read( $D$ )

**end**



# Automatic Acquisition of Locks: Write

- **write( $D$ )** is processed as:

```
if  $T_i$  has a lock-X on  $D$ 
  then
    write( $D$ )
  else begin
    if necessary wait until no other transaction has any lock on  $D$ ,
    if  $T_i$  has a lock-S on  $D$ 
      then
        upgrade lock on  $D$  to lock-X
      else
        grant  $T_i$  a lock-X on  $D$ 
    write( $D$ )
  end;
```

- All locks are released after commit or abort



# Deadlocks

- Two-phase locking *does not* ensure freedom from deadlocks

$T_3$ :

```
lock-X( $B$ );
read( $B$ );
 $B := B - 50$ ;
write( $B$ );
lock-X( $A$ );
read( $A$ );
 $A := A + 50$ ;
write( $A$ );
unlock( $B$ );
unlock( $A$ )
```

$T_4$ :

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

$T_3$	$T_4$
lock-x ( $B$ ) read ( $B$ ) $B := B - 50$ write ( $B$ )	lock-s ( $A$ ) read ( $A$ ) lock-s ( $B$ )
lock-x ( $A$ )	

- Observe that transactions  $T_3$  and  $T_4$  are two phase, but, in deadlock



# Starvation

- In addition to deadlocks, there is a possibility of **starvation**
- **Starvation** occurs if the concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item
  - The same transaction is repeatedly rolled back due to deadlocks
- Concurrency control manager can be designed to prevent starvation



# Cascading roll-back

- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil
- When a deadlock occurs there is a possibility of cascading roll-backs
- Cascading roll-back is possible under two-phase locking
- In the schedule here, each transaction observes the two-phase locking protocol, but the failure of  $T_5$  after the  $\text{read}(A)$  step of  $T_7$  leads to cascading rollback of  $T_6$  and  $T_7$ .

$T_5$	$T_6$	$T_7$
lock-X( $A$ ) read( $A$ ) lock-S( $B$ ) read( $B$ ) write( $A$ ) unlock( $A$ )		
	lock-x( $A$ ) read( $A$ ) write( $A$ ) unlock( $A$ )	
		lock-S( $A$ ) read( $A$ )



# More Two Phase Locking Protocols

- To avoid Cascading roll-back, follow a modified protocol called **strict two-phase locking**
  - a transaction must hold all its exclusive locks till it commits/aborts
- **Rigorous two-phase locking** is even stricter.
  - *All* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit

NPTEL



- Concurrency Control
- Lock-Based Protocols
- **Implementing Locking**

NPTEL

# IMPLEMENTING LOCKING

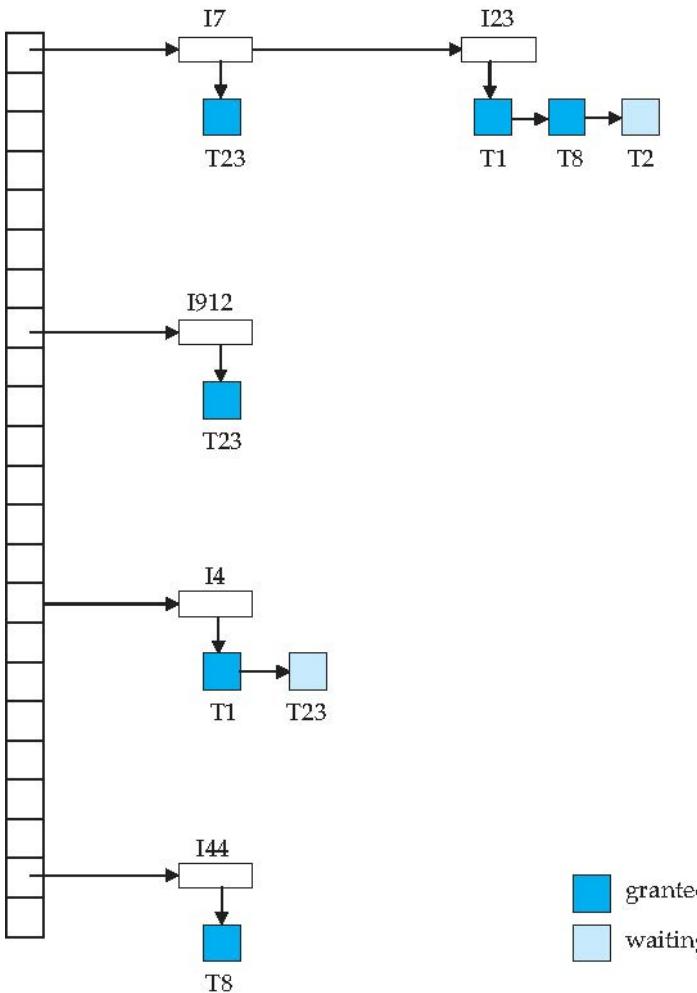


# Implementation of Locking

- A **lock manager** can be implemented as a separate process to which transactions send lock and unlock requests
- The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock)
- The requesting transaction waits until its request is answered
- The lock manager maintains a data-structure called a **lock table** to record granted locks and pending requests
- The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked



# Lock Table



- Dark blue rectangles indicate granted locks; light blue indicate waiting requests
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted
  - lock manager may keep a list of locks held by each transaction, to implement this efficiently



# Module Summary

- Understood the locking mechanism and protocols
- Realized that deadlock is a peril of locking and needs to be handled through rollback

NPTEL



# Instructor and TAs

Name	Mail	Mobile
Partha Pratim Das, Instructor	ppd@cse.iitkgp.ernet.in	9830030880
Srijoni Majumdar, TA	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, TA	himadribhuyan@gmail.com	9438911655
Gurunath Reddy M	mgurunathreddy@gmail.com	9434137638

**Slides used in this presentation are borrowed from <http://db-book.com/> with kind permission of the authors.**

**Edited and new slides are marked with “PPD”.**



# Database Management Systems

## Module 35: Concurrency Control/2

**Partha Pratim Das**

*Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur*

[ppd@cse.iitkgp.ernet.in](mailto:ppd@cse.iitkgp.ernet.in)

**Srijoni Majumdar  
Himadri B G S Bhuyan  
Gurunath Reddy M**



**Database System Concepts, 6<sup>th</sup> Ed.**

©Silberschatz, Korth and Sudarshan  
[www.db-book.com](http://www.db-book.com)



# Module Recap

- Concurrency Control
- Lock-Based Protocols
- Implementing Locking

NPTEL



# Module Objectives

- Deadlocks are peril of locking. We need to understand how to detect, prevent and recover from deadlock
- Introduce a simple time-based protocol that avoids deadlocks

NPTEL



# Module Outline

- Deadlock Handling
- Timestamp-Based Protocols

NPTEL



- **Deadlock Handling**
- Timestamp-Based Protocols

NPTEL

# DEADLOCK HANDLING



# 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
- **Deadlock prevention** protocols ensure that the system will *never* enter into a deadlock state.  
Some prevention strategies :
  - Require that each transaction locks all its data items before it begins execution (predeclaration)
  - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order



# Deadlock Prevention

- Following schemes use transaction timestamps for the sake of deadlock prevention alone
- **wait-die** scheme — non-preemptive
  - Older transaction may wait for younger one to release data item. (older means smaller timestamp)
    - ▶ Younger transactions never wait for older ones; they are rolled back instead
  - A transaction may die several times before acquiring needed data item
- **wound-wait** scheme — preemptive
  - Older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it
    - ▶ Younger transactions may wait for older ones
  - May be fewer rollbacks than *wait-die* scheme



# Deadlock Prevention

- Both in *wait-die* and in *wound-wait* schemes, a rolled back transaction is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided
- **Timeout-Based Schemes:**
  - a transaction waits for a lock only for a specified amount of time. If the lock has not been granted within that time, the transaction is rolled back and restarted,
  - Thus, deadlocks are not possible
  - simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval

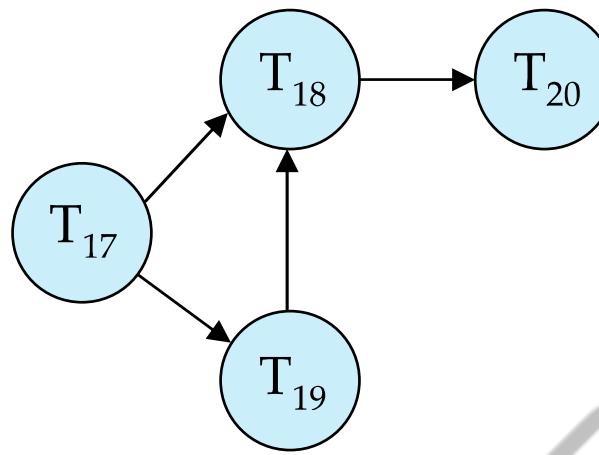


# Deadlock Detection

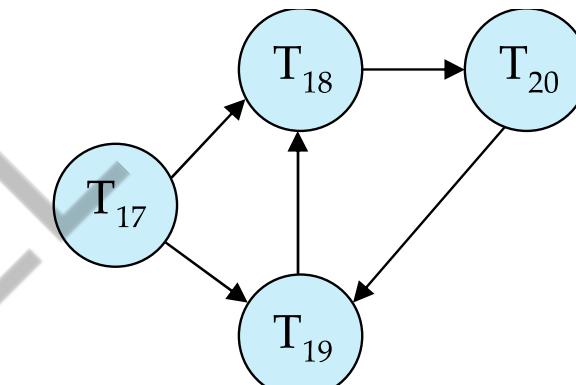
- Deadlocks can be described as a *wait-for graph*, which consists of a pair  $G = (V, E)$ ,
  - $V$  is a set of vertices (all the transactions in the system)
  - $E$  is a set of edges; each element is an ordered pair  $T_i \rightarrow T_j$ .
- If  $T_i \rightarrow T_j$  is in  $E$ , then there is a directed edge from  $T_i$  to  $T_j$ , implying that  $T_i$  is waiting for  $T_j$  to release a data item
- When  $T_i$  requests a data item currently being held by  $T_j$ , then the edge  $T_i \rightarrow T_j$  is inserted in the wait-for graph. This edge is removed only when  $T_j$  is no longer holding a data item needed by  $T_i$
- The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles



# Deadlock Detection: Example



Wait-for graph without a cycle



Wait-for graph with a cycle



# Deadlock Recovery

- When deadlock is detected :

- Some transaction will have to rolled back (made a victim) to break deadlock. Select that transaction as victim that will incur minimum cost
- Rollback -- determine how far to roll back transaction
  - ▶ **Total rollback:** Abort the transaction and then restart it
  - ▶ More effective to roll back transaction only as far as necessary to break deadlock
- Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation



- Deadlock Handling
- **Timestamp-Based Protocols**

NPTEL

# TIMESTAMP-BASED PROTOCOLS



# Timestamp-Based Protocols

- Each transaction is issued a timestamp when it enters the system. If an old transaction  $T_i$  has time-stamp  $\text{TS}(T_i)$ , a new transaction  $T_j$  is assigned time-stamp  $\text{TS}(T_j)$  such that  $\text{TS}(T_i) < \text{TS}(T_j)$ .
- The protocol manages concurrent execution such that the time-stamps determine the serializability order
- In order to assure such behavior, the protocol maintains for each data  $Q$  two timestamp values:
  - **W-timestamp( $Q$ )** is the largest time-stamp of any transaction that executed **write( $Q$ )** successfully
  - **R-timestamp( $Q$ )** is the largest time-stamp of any transaction that executed **read( $Q$ )** successfully



# Timestamp-Based Protocols

- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order
- Suppose a transaction  $T_i$  issues a **read**( $Q$ )
  1. If  $TS(T_i) \leq W\text{-timestamp}(Q)$ , then  $T_i$  needs to read a value of  $Q$  that was already overwritten.
    - Hence, the **read** operation is rejected, and  $T_i$  is rolled back.
  2. If  $TS(T_i) \geq W\text{-timestamp}(Q)$ , then the **read** operation is executed, and **R**-timestamp( $Q$ ) is set to **max**(**R**-timestamp( $Q$ ),  $TS(T_i)$ ).



# Timestamp-Based Protocols (Cont.)

- Suppose that transaction  $T_i$  issues **write**(Q).
  1. If  $TS(T_i) < \mathbf{R}\text{-timestamp}(Q)$ , then the value of Q that  $T_i$  is producing was needed previously, and the system assumed that that value would never be produced
    - Hence, the **write** operation is rejected, and  $T_i$  is rolled back
  2. If  $TS(T_i) < \mathbf{W}\text{-timestamp}(Q)$ , then  $T_i$  is attempting to write an obsolete value of Q
    - Hence, this **write** operation is rejected, and  $T_i$  is rolled back
  3. Otherwise, the **write** operation is executed, and  $\mathbf{W}\text{-timestamp}(Q)$  is set to  $TS(T_i)$



# Example Use of the Protocol

A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5

$T_1$	$T_2$	$T_3$	$T_4$	$T_5$
read ( $Y$ )	read ( $Y$ )	write ( $Y$ ) write ( $Z$ )	read ( $W$ )	read ( $X$ )
read ( $X$ )	read ( $Z$ ) abort	write ( $W$ ) abort		read ( $Z$ )
				write ( $Y$ ) write ( $Z$ )



# Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:



Thus, there will be no cycles in the precedence graph

- Timestamp protocol ensures freedom from deadlock as no transaction ever waits
- But the schedule may not be cascade-free, and may not even be recoverable



# Module Summary

- Explained how to detect, prevent and recover from deadlock
- Introduced a time-based protocol that avoids deadlocks

NPTEL



# Instructor and TAs

Name	Mail	Mobile
Partha Pratim Das, Instructor	ppd@cse.iitkgp.ernet.in	9830030880
Srijoni Majumdar, TA	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, TA	himadribhuyan@gmail.com	9438911655
Gurunath Reddy M	mgurunathreddy@gmail.com	9434137638

**Slides used in this presentation are borrowed from <http://db-book.com/> with kind permission of the authors.**

**Edited and new slides are marked with “PPD”.**