Database Systems
(CS 355 / CE 373)

Dr. Umer Tariq
Assistant Professor,
Dhanani School of Science & Engineering,
Habib University

Acknowledgements

- Many slides have been borrowed from the official lecture slides accompanying the textbook:

  Database System Concepts, (2019), Seventh Edition,

  Avi Silberschatz, Henry F. Korth, S. Sudarshan

  McGraw-Hill, ISBN 9780078022159

  The original lecture slides are available at:

  https://www.db-book.com/

- Some of the slides have been borrowed from the lectures by Dr. Immanuel Trummer (Cornell University). Available at: (www.itrummer.org)

Outline: Week 11

- Transactions: Deeper Dive into Isolation Property/Requirement
- Transaction Schedules: Serial vs Concurrent
- Equivalent Schedules
- Serializable Schedules
- Testing for Serializability

DBMS: The Concept of *Transactions*

DB

Accounts

Number	Balance	C-ID

Account(C-ID, Number, Balance)

Application Code.

TRANSFER

OnButtonClicked OK

SELECT Balance
FROM Accounts
WHERE Number = '.... 1245' Read(A)
=> BalanceVar

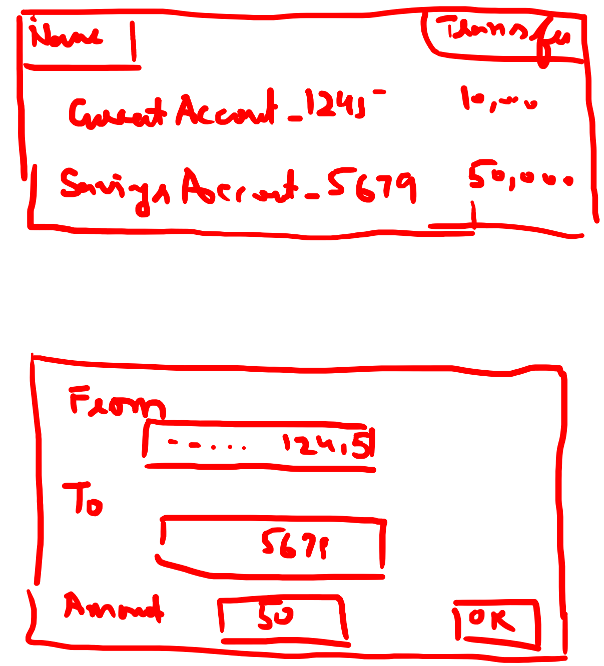
BalanceVar = BalanceVar - 50

Update Accounts Write(A)
Set Balance = BalanceVar
WHERE Number = '....1245';

SELECT Balance
FROM Accounts Read(B)
WHERE Number = '....5679'
=> BalanceVar
Balance = BalanceVar + 50

Update Accounts Write(B)
Set Balance = BalanceVar
WHERE Number = '....5679'

UI



DBMS: The Concept of *Transactions*

TRANSFER (Transfer. Prg 1

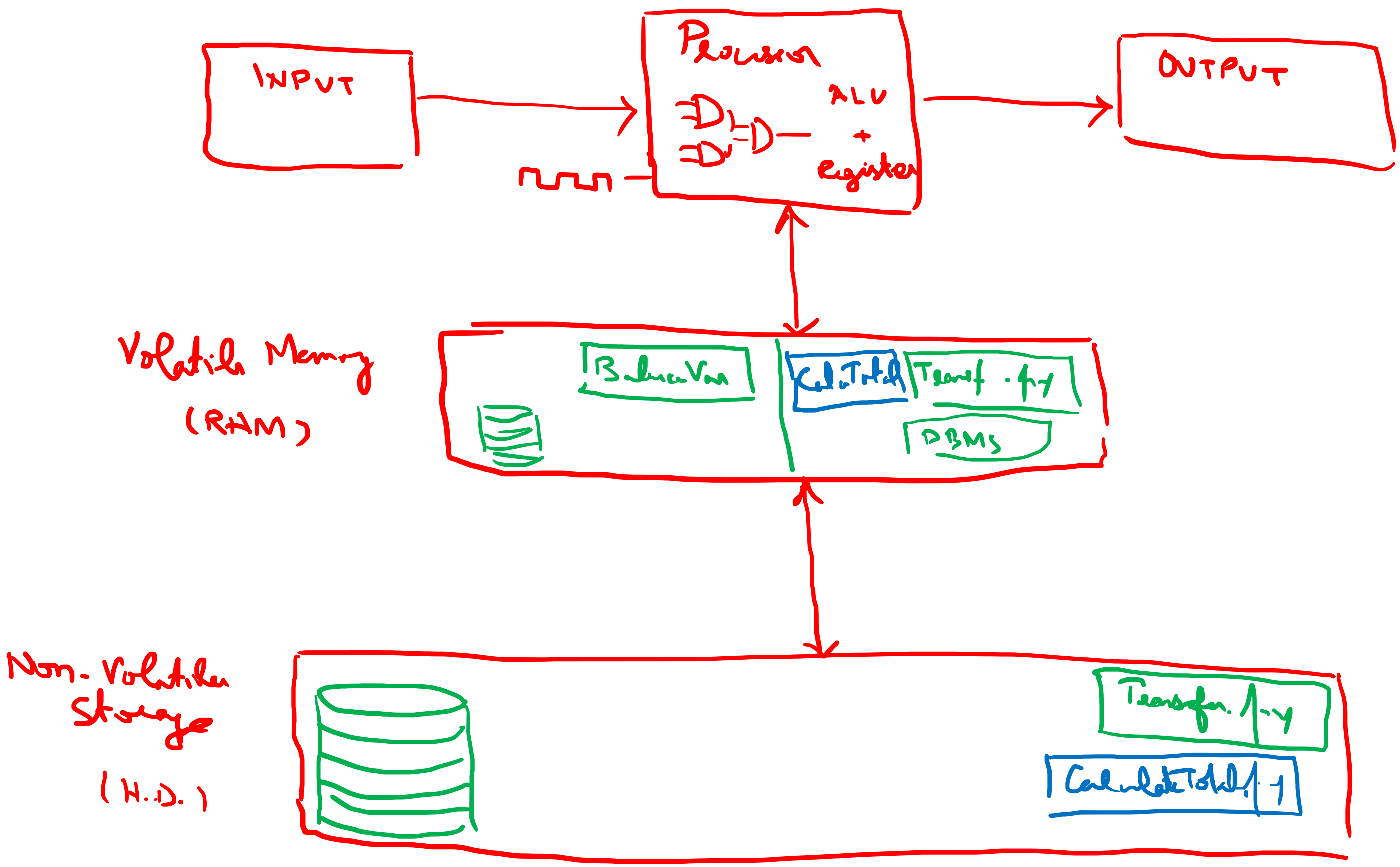
BEGIN TRANSACTION

1. Read (A)
2. A = A − 50
3. Write (A)
4. Read (B)
5. B = B + 50
6. Write (B)

END TRANSACTION

DBMS: The Concept of ***Transactions***

Draw a "COMPUTER"?



DBMS: The Concept of *Transactions*

- Transaction
  - A transaction is unit of program execution that consists of multiple database operations but appears as a single, indivisible unit from the point of view of the database user/application.
  - A transaction executes in its entirety or not at all.

- Example
  - A transaction to transfer Rs. 50 from account A to account B

T_i: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

Transaction Properties: ACID

- A transaction must have the following four properties:
  - **A**tomicity,
  - **C**onsistency,
  - **I**solation,
  - **D**urability.

- These form the acronym **ACID** properties.

DIVE DEEP INTO "ISOLATION" Property

Serial vs Concurrent Execution of Transactions

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

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

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

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

Serial vs Concurrent Execution of Transactions

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

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

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

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

- Restricting ourselves to executing transactions serially (i.e. one after the other) makes it easy to achieve isolation among transactions.
- However, concurrent execution of transactions provides significant performance benefits:
  - Increased throughput
  - Reduced average response times

Concurrent Execution of Transactions: Role of Concurrency-Control Schemes

- Concurrency-control schemes
  - Mechanisms to achieve isolation among concurrently-executing transactions
  - Mechanisms to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

- Will study these schemes after studying the notion of ‘correctness of concurrent executions’

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

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

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

Transaction Schedule

- When transactions are executing concurrently in an interleaved fashion, then the order of execution of operations from all the various transactions is known as a **schedule** (or **history**).

	T_1
	read(A)
	$A := A - 50$
2,	write(A)
	read(B)
1,	$B := B + 50$
	write(B)
	commit

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

S1

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

S2

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	

S3

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

S4

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

Transaction "Schedule"

- A **schedule** (or **history**) *S* of *n* transactions $T_1, T_2, \ldots, T_n$ is an ordering of the operations of the transactions.

- Operations from different transactions can be interleaved in the schedule *S*.

- However, for each transaction T_i that participates in the schedule *S*, the operations of T_i in *S* must appear in the same order in which they occur in T_i.

	T_1
	read(A)
	$A := A - 50$
2)	write(A)
	read(B)
1)	$B := B + 50$
	write(B)
	commit

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

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

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

Serial Schedule

- A **serial schedule** consists of a sequence of instructions from various transactions, where the instructions belonging to one single transaction appear together in that schedule.

- How many serial schedules are possible?

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

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

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

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	

T_1 : A = 50 B = 150 T_2 : A = 45 B = 155 | T_2 : A = 90 B = 110 T_1 : A = 40 , B = 160

Concurrent Schedule: Consistent vs Inconsistent State

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

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

A = 100   B = 100

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

A = 45   B = 155

Var A_1 = 100
Var A_1 = 50
DB - A = 50
Var A = 50
$temp^2$ = 5
Var A_2 = 45
DB - A = 45
Var B_1 = 100
Var B_1 = 150
DB - B = 150
Var B_2 = 150
Var B_2 = 155
DB - B = 155

Var A_1 = 100
Var A_1 = 50
Var A_2 = 100
temp = 10
Var A_2 = 90
DB - A = 90
Var B_2 = 100
DB - A = 50
Var B_1 = 100
Var B_1 = 150
DB - B = 150
Var B_2 = 110
DB - B = 110

A = 100 , B = 100

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 = 50   B = 110

Serializable Schedule

- A serial schedule is always consistent, i.e. it maintains the consistent state of the database.

- However, the same cannot be guaranteed for a concurrent schedule.

- If a concurrent schedule can be shown to have the same effect as a serial schedule, (in other words it is shown to be equivalent to a serial schedule), then it can ensure the consistency of the database.

- Such a concurrent schedule is called a **serializable schedule.**

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

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

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

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

Serializable Schedules: Equivalence

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

Figure 17.4 Schedule 3—a concurrent schedule equivalent to schedule 1.

E Q U I V A L E N T T O

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

Figure 17.2 Schedule 1—a serial schedule in which T_1 is followed by T_2.

<u>If a concurrent schedule can be shown to be equivalent to a serial schedule, we conclude that this concurrent schedule maintains the consistency of the database.</u>

How Can We Evaluate Schedule Equivalence: Swapping

- Let *I* and *J* be consecutive instructions of a schedule *S*.
- If *I* and *J* are instructions of different transactions and *I* and *J* do not conflict, then we can swap the order of *I* and *J* to produce a new schedule *S*′.
- *S* is equivalent to *S*′, since all instructions appear in the same order in both schedules except for *I* and *J*, whose order does not matter.
- We can swap the instructions *read*(*B*) and *write*(*A*) as they do not conflict.

S_1

T_1	T_2
read(A)	
write(A)	
	read(A)
	$\boxed{\text{write}(A)}$
$\boxed{\text{read}(B)}$	
write(B)	
	read(B)
	write(B)

→

S_2

T_1	T_2
read(A)	
write(A)	
	read(A)
$\boxed{\text{read}(B)}$	
	$\boxed{\text{write}(A)}$
write(B)	
	read(B)
	write(B)

How Can We Evaluate Schedule Equivalence: Swapping

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

Figure 17.4 Schedule 3—a concurrent schedule equivalent to schedule 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

Figure 17.2 Schedule 1—a serial schedule in which T_1 is followed by T_2.

<u>If a concurrent schedule can be shown to be equivalent to a serial schedule, we conclude that this concurrent schedule maintains the consistency of the database.</u>

Transaction Schedules: Equivalence

- We consider only ***read(X)*** and ***write(X)*** operations. The underlying assumption is that these two operations are the most significant operations from a scheduling perspective.

- Lets consider a schedule *S* in which there are two consecutive instructions *I* and *J* of transactions *Ti* and *Tj* respectively.

- If *I* and *J* refer to different data items, then we can swap *I* and *J* without affecting the results of instructions in the schedule.

T1	T2
write ~~*read*~~(*X*)	
	write ~~*read*~~(*Y*)

Initial (Above), Swapped (Below)

T1	T2
	write ~~*read*~~(*Y*)
write ~~*read*~~(*X*)	

Transaction Schedules: Equivalence

- Let's consider a schedule *S* in which there are two consecutive instructions *I* and *J* of transactions *Ti* and *Tj* respectively.

- if *I* and *J* refer to the same data item *Q*, then the order of the two steps may matter.
  - Case1: *I* = *read*(*Q*), *J* = *read*(*Q*).
  - Case2: *I* = *read*(*Q*), *J* = *write*(*Q*).
  - Case3: *I* = *write*(*Q*), *J* = *read*(*Q*).
  - Case4: *I* = *write*(*Q*), *J* = *write*(*Q*).

	read(Q)
read(Q)	

T1	T2
read(*Q*)	
	read(*Q*)

Case1

	write(Q)
read(Q)	

T1	T2
read(*Q*)	
	write(*Q*)

Case2

	read(Q)
write(Q)	

T1	T2
write(*Q*)	
	read(*Q*)

Case3

	write(Q)
write(Q)	

T1	T2
write(*Q*)	
	write(*Q*)

Case4

Transaction Schedules: Equivalence

- Case1: *I = read(Q), J = read(Q)*
  - The order of *I* and *J* does not matter, since the same value of *Q* is read by T_i and T_j, regardless of the order.

T1	T2
read(Q)	
	read(Q)

Initial (Left),
Swapped (Right)
Equivalent

T1	T2
	read(Q)
read(Q)	

Transaction Schedules: Equivalence

- Case2: *I = read(Q), J = write(Q)*
  - If *I* comes before *J*, then T_i does not read the value of *Q* that is written by T_j in instruction *J*.
  - If *J* comes before *I*, then T_i reads the value of *Q* that is written by T_j.
  - Thus, the order of *I* and *J* matters.

T1	T2
read(Q)	
	write(Q)

Initial (Left),
Swapped (Right)
Not Equivalent

T1	T2
	write(Q)
read(Q)	

Transaction Schedules: Equivalence

- Case3: *I = write(Q), J = read(Q)*
  - The order of *I* and *J* matters for reasons similar to those of the previous case.

T1	T2
write(Q)	
	read(Q)

Initial (Left),
Swapped (Right)
Not Equivalent

T1	T2
	read(Q)
write(Q)	

Transaction Schedules: Equivalence

- Case4: $I = write(Q)$, $J = write(Q)$
  - Since both instructions are write operations, the order of these instructions does not affect either T_i or T_j
  - However, the value obtained by the next *read*(Q) instruction of S is affected, since the result of only the latter of the two write instructions is preserved in the database.
  - If there is no other *write*(Q) instruction after I and J in S, then the order of I and J directly affects the final value of Q in the database state that results from schedule S.

T1	T2
write(Q)	
	write(Q)

Initial (Left),
Swapped (Right)
Not Equivalent

T1	T2
	write(Q)
write(Q)	

Transaction Schedules: Conflict

- Thus, only in the case where both *I* and *J* are read instructions does the relative order of their execution not matter.

- We say that *I* and *J* conflict if they are operations by different transactions on the same data item, and **at least one of these instructions is a write operation**.

<table>
<tr><th>Initial Schedule</th><th>Swapped Schedule</th><th>Conflict?</th></tr>
<tr><td><table><tr><th>T1</th><th>T2</th></tr><tr><td>read(Q)</td><td></td></tr><tr><td></td><td>read(Q)</td></tr></table></td><td><table><tr><th>T1</th><th>T2</th></tr><tr><td></td><td>read(Q)</td></tr><tr><td>read(Q)</td><td></td></tr></table></td><td>No</td></tr>
<tr><td><table><tr><th>T1</th><th>T2</th></tr><tr><td>read(Q)</td><td></td></tr><tr><td></td><td>write(Q)</td></tr></table></td><td><table><tr><th>T1</th><th>T2</th></tr><tr><td></td><td>write(Q)</td></tr><tr><td>read(Q)</td><td></td></tr></table></td><td>Yes</td></tr>
<tr><td><table><tr><th>T1</th><th>T2</th></tr><tr><td>write(Q)</td><td></td></tr><tr><td></td><td>read(Q)</td></tr></table></td><td><table><tr><th>T1</th><th>T2</th></tr><tr><td></td><td>read(Q)</td></tr><tr><td>write(Q)</td><td></td></tr></table></td><td>Yes</td></tr>
<tr><td><table><tr><th>T1</th><th>T2</th></tr><tr><td>write(Q)</td><td></td></tr><tr><td></td><td>write(Q)</td></tr></table></td><td><table><tr><th>T1</th><th>T2</th></tr><tr><td></td><td>write(Q)</td></tr><tr><td>write(Q)</td><td></td></tr></table></td><td>Yes</td></tr>
</table>

Transaction Schedules: Swapping

- Let *I* and *J* be consecutive instructions of a schedule *S*.
- If *I* and *J* are instructions of different transactions and *I* and *J* do not conflict, then we can swap the order of *I* and *J* to produce a new schedule *S*′.
- *S* is equivalent to *S*′, since all instructions appear in the same order in both schedules except for *I* and *J*, whose order does not matter.
- We can swap the instructions *read*(*B*) and *write*(*A*) as they do not conflict.

S_1

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

→

S_2

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

Transaction Schedules: Swapping

- We can again swap the instructions *read*(*B*) and *read*(*A*) as they do not conflict.

S_2

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

$\rightarrow$

S_3

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

Transaction Schedules: Swapping

- We can now swap the instructions *write*(*B*) and *write*(*A*) as they do not conflict.

S_3

T_1	T_2
read(A)	
write(A)	
read(B)	
	read(A)
	$\boxed{\text{write}(A)}$
$\boxed{\text{write}(B)}$	
	read(B)
	write(B)

→

S_4

T_1	T_2
read(A)	
write(A)	
read(B)	
	read(A)
$\boxed{\text{write}(B)}$	
	$\boxed{\text{write}(A)}$
	read(B)
	write(B)

Transaction Schedules: Swapping

- Finally, we can swap the instructions *write*(*B*) and *read*(*A*) as they do not conflict.

S_4

T_1	T_2
read(A)	
write(A)	
read(B)	
	$\boxed{\text{read}(A)}$
$\boxed{\text{write}(B)}$	
	write(A)
	read(B)
	write(B)

→

S_5

T_1	T_2
read(A)	
write(A)	
read(B)	
$\boxed{\text{write}(B)}$	
	$\boxed{\text{read}(A)}$
	write(A)
	read(B)
	write(B)

✓

$S_1 = S_2 = S_3 = S_4 = S_5$

Serial

Transaction Schedules: Swapping

- Note that the final schedule (after the various swapping steps) is a serial schedule.

- Since the original schedule has been shown to be equivalent to a serial schedule, we conclude that the original schedule maintains the consistency of the database.

S_1

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

EQUIVALENT TO

S_1 —

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

Conflict-Equivalent Schedules

- If a schedule S can be transformed into a schedule S′ by a series of swaps of nonconflicting instructions, we say that S and S′ are **conflict equivalent.**

S

S_1

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

EQUIVALENT TO

S'

S_{1}–

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

Conflict Serializability

- A schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule.
- Consider the following schedule:

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

- This schedule is not conflict serializable as it is not equivalent to the serial schedule <T_3, T_4> or the serial schedule <T_4, T_3>.

Conflict Serializability

- A schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule.
- Consider the following schedule:

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

Exercise

17.15 Consider the following two transactions:

T_{13}: read(A);
read(B);
if $A = 0$ **then** $B := B + 1$;
write(B).

T_{14}: read(B);
read(A);
if $B = 0$ **then** $A := A + 1$;
write(A).

Let the consistency requirement be $A = 0 \vee B = 0$, with $A = B = 0$ as the initial values.

a. Show that every serial execution involving these two transactions preserves the consistency of the database.

b. Show a concurrent execution of T_{13} and T_{14} that produces a nonserializable schedule.

c. Is there a concurrent execution of T_{13} and T_{14} that produces a serializable schedule?

Exercise

17.15 Consider the following two transactions:

T_{13}: read(A);	T_{14}: read(B);
read(B);	read(A);
if $A = 0$ **then** $B := B + 1$;	**if** $B = 0$ **then** $A := A + 1$;
write(B).	write(A).

Let the consistency requirement be $A = 0 \lor B = 0$, with $A = B = 0$ as the initial values.

a. Show that every serial execution involving these two transactions preserves the "~~consistency~~" ~~of the database~~. the consistency required

<T_{13}, T_{14}>

A=0 B=

read(A) var A_1=0

read(B) var B_1=0

if A=0 ...

write(B)

read(B)

read(A)

<T_{14}, T_{13}>

Exercise

17.15 Consider the following two transactions:

T_{13}: read(A);
read(B);
if $A = 0$ **then** $B := B + 1$;
write(B).

T_{14}: read(B);
read(A);
if $B = 0$ **then** $A := A + 1$;
write(A).

(schedule)

b. Show a concurrent execution of T_{13} and T_{14} that produces a nonserializable schedule.

read(A)	
read(B)	
	read(B)
	read(A)
if A:=0 then B:=B+1	
	if B:=0 then A:A+1
write(B)	
	write(A)

Testing for Conflict Serializability: Precedence Graphs

concurrent

- To find whether a schedule is **conflict serializable**, we use precedence graphs.

Testing for Conflict Serializability: Precedence Graphs

- To find whether a schedule is **conflict serializable**, we use precedence graphs.

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

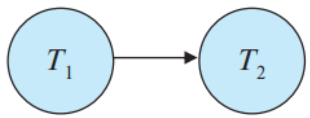
T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	$B := B + temp$
	write(B)
	commit

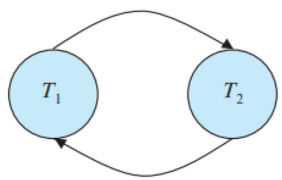
Testing for Conflict Serializability: Precedence Graphs

- To find whether a schedule is **conflict serializable**, we use precedence graphs.

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

Precedence Graph of the (left) schedule





Precedence Graph of the (right) schedule

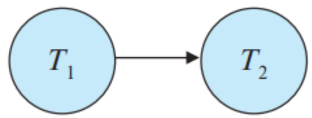
T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	$B := B + temp$
	write(B)
	commit

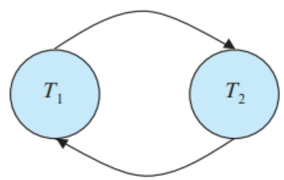
"Testing for Conflict Serializability": Precedence Graphs

- To find whether a schedule is **conflict serializable**, we use precedence graphs.

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

Precedence Graph of the (left) schedule





Precedence Graph of the (right) schedule

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 schedule is conflict serializable if and only if its precedence graph is acyclic.

Transaction Schedules: How to Build Precedence Graphs

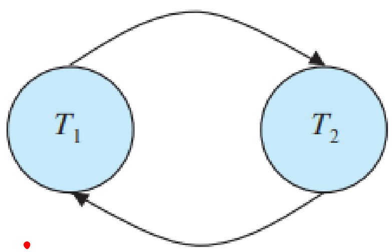
- Precedence graph $G = (V, E)$, has V as its set of vertices and E as its set of edges.
  - The set of vertices consists of all the transactions participating in the schedule.
  - The set of edges consists of all edges $T_i \rightarrow T_j$ for which one of three conditions holds:
    - T_i executes *write*(Q) before T_j executes *read*(Q).
    - T_i executes *read*(Q) before T_j executes *write*(Q).
    - T_i executes *write*(Q) before T_j executes *write*(Q).

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

Transaction Schedules: How to Build Precedence Graphs

- Precedence graph $G = (V, E)$, has V as its set of vertices and E as its set of edges.
  - The set of vertices consists of all the transactions participating in the schedule.
  - The set of edges consists of all edges $T_i \rightarrow T_j$ for which one of three conditions holds:
    - T_i executes *write*(Q) before T_j executes *read*(Q).
    - T_i executes *read*(Q) before T_j executes *write*(Q).
    - T_i executes *write*(Q) before T_j executes *write*(Q).

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



Rationale for the Test :

If an edge $T_i \rightarrow T_j$ exists in the precedence graph, then, in any serial schedule S' equivalent to S, T_i must appear before T_j.

T_1 | T_2 | T_3 | T_4

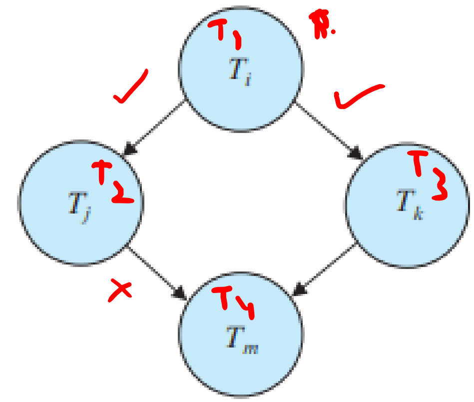
$\langle T_1, T_2, T_3, T_4 \rangle$ $\langle T_4, T_3, T_2, T_1 \rangle$

$\langle T_2, T_1, T_3, T_4 \rangle$ $\langle T_4, T_1, T_2, T_3 \rangle$

Serializability Order

Precedence Graph

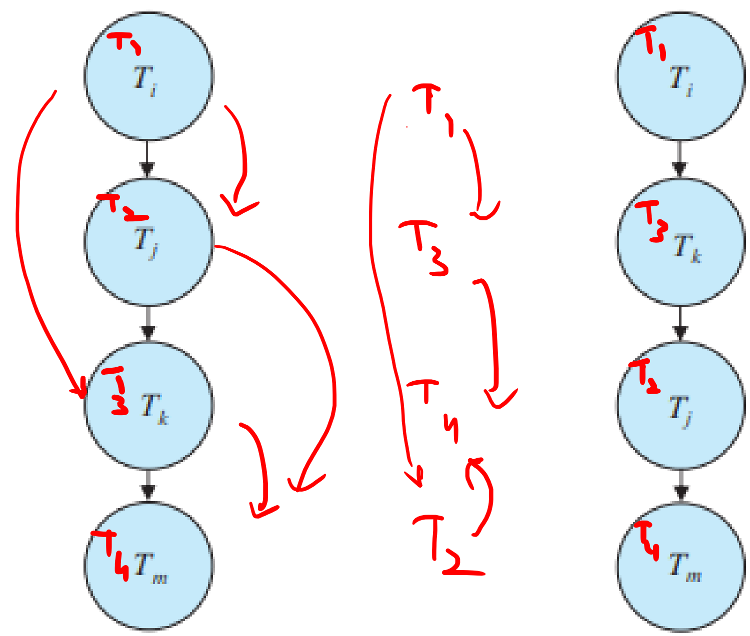
- If the precedence graph of a transaction schedule is acyclic, a serializability order of the transactions can be obtained by **topological sorting** of the precedence graph.



- Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge u→v, vertex u comes before v in the ordering

$\langle T_1, T_2, T_3, T_4 \rangle$

- There are, in general, several possible linear orders that can be obtained through a topological sort.



Testing for Conflict Serializability: Summary

concurrent

- To test whether a schedule is "**conflict serializable**," we need to
  1. construct the precedence graph
  2. invoke a cycle-detection algorithm.

- 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.

- To determine a "serializability order" from a precedence graph, algorithms can also be invoked to find topological sort of vertices of a directed acyclic graph (DAG).

Exercise

17.6 Consider the precedence graph of Figure 17.16. Is the corresponding schedule conflict serializable? Explain your answer.

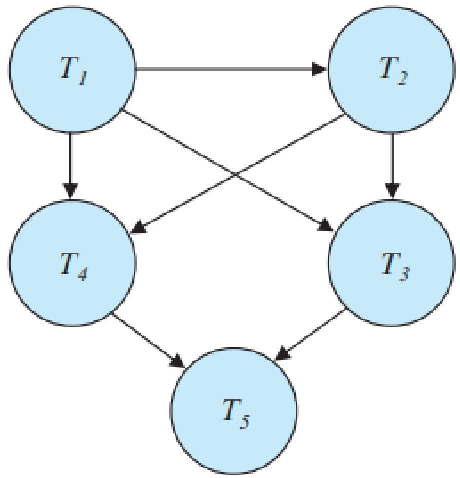


Figure 17.16 Precedence graph for Practice Exercise 17.6.

Exercise

17.15 Consider the following two transactions:

T_{13}: read(A);	T_{14}: read(B);
read(B);	read(A);
if $A = 0$ **then** $B := B + 1$;	**if** $B = 0$ **then** $A := A + 1$;
write(B).	write(A).

schedule

c. Is there a concurrent execution of T_{13} and T_{14} that produces a serializable schedule?

T_{13}	T_{14}
read(A)	
	read(B)
read(B)	
	read(A)
if --	
	if --
write(B)	
	write(A)

$$\begin{tikzcd} T_{13} \arrow[r, bend left] & T_{14} \arrow[l, bend left] \end{tikzcd}$$

T_1 , T_5

A = 50
B = 150

A = 60
B = 140

"Beyond Conflict Serializability" Other Notions of Serializability

- Sometimes a schedule may not be conflict serializable, however it may produce the same view as a serial schedule.
  - In other words, a schedule's precedence graph may have cycles, however it may give a consistent result.

$T_1 \rightleftarrows T_5$

A = 100 , B = 100

T_1	T_5
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)

var A_1 = 100
var A_1 = 50
DB_A = 50!
var B_5 = 100
var B_5 = 90
DB_B = 90!
var B_1 = 90
var B_1 = 140
$\boxed{\text{DB\_B = 140!}}$
var A_5 = 50
var A_5 = 60
$\boxed{\text{DB\_A = 60!}}$

A = 140 B = 60

Beyond Conflict Serializability: Other Notions of Serializability

- Sometimes a schedule may not be conflict serializable, however it may produce the same view as a serial schedule.
  - In other words, a schedule's precedence graph may have cycles, however it may give a consistent result.

T_1	T_5
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)

- The schedule above produces same outcome as the serial schedule < T1, T5 >, yet is not conflict equivalent