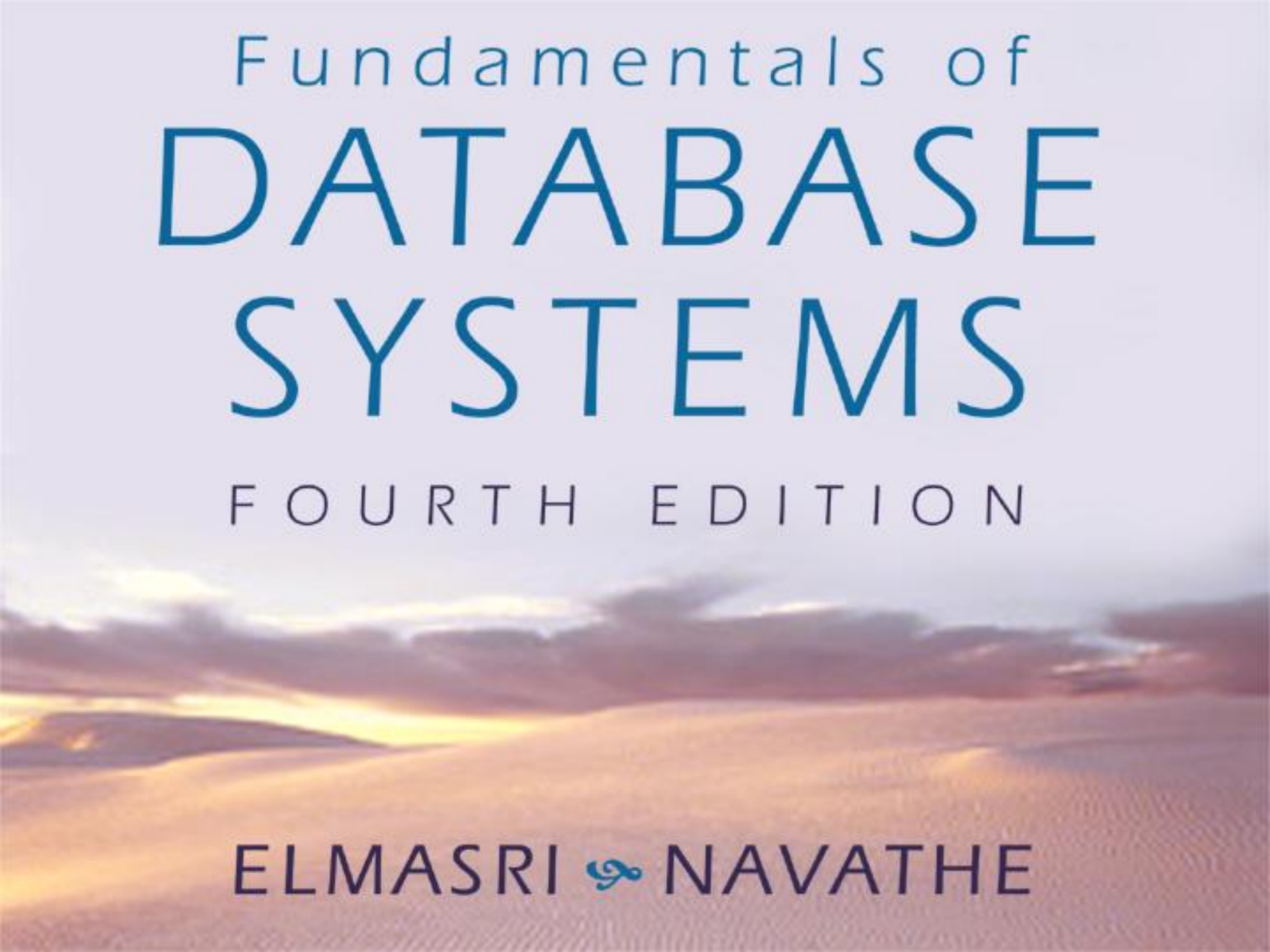
Fundamentals of DATABASE SYSTEMS

FOURTH EDITION



ELMASRI NAVATHE

Chapter 17

Introduction to Transaction Processing Concepts and Theory

Chapter Outline

1 Introduction to Transaction Processing

2 Transaction and System Concepts

3 Desirable Properties of Transactions

4 Characterizing Schedules based on Recoverability

5 Characterizing Schedules based on Serializability

6 Transaction Support in SQL

1 Introduction to Transaction Processing (1)

- **Single-User System:** At most one user at a time can use the system.
- **Multiuser System**: Many users can access the system concurrently.
- **Concurrency**
  - **Interleaved processing**: concurrent execution of processes is interleaved in a single CPU
  - **Parallel processing**: processes are concurrently executed in multiple CPUs.

Introduction to Transaction Processing (2)

- **A Transaction:** logical unit of database processing that includes one or more access operations (read -retrieval, write - insert or update, delete).
- **A transaction (set of operations)** may be stand-alone specified in a high level language like SQL submitted interactively, or may be embedded within a program.
- **Transaction boundaries**: Begin and End transaction.
- An **application program** may contain several transactions separated by the Begin and End transaction boundaries.

Introduction to Transaction Processing (3)

SIMPLE MODEL OF A DATABASE (for purposes of discussing transactions):

- **A database -** collection of named data items
- **Granularity of data** - a field(attribute), a record(tuple), or a whole disk block (Concepts are independent of granularity)
- Basic operations are **read** and **write**
  - **read_item(X)**: Reads a database item named X into a program variable. To simplify our notation, we assume that *the program variable is also named X*.
  - **write_item(X)**: Writes the value of program variable X into the database item named X.

Introduction to Transaction Processing (4)

READ AND WRITE OPERATIONS:

- Basic unit of data transfer from the disk to the computer main memory is one block. In general, a data item (what is read or written) will be the field of some record in the database, although it may be a larger unit such as a record or even a whole block.
- **read_item(X) command includes the following steps:**

1. Find the address of the disk block that contains item X.
2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
3. Copy item X from the buffer to the program variable named X.

Introduction to Transaction Processing (5)

READ AND WRITE OPERATIONS (cont.):

- **write_item(X) command includes the following steps:**

1. Find the address of the disk block that contains item X.
2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
3. Copy item X from the program variable named X into its correct location in the buffer.
4. Store the updated block from the buffer back to disk (either immediately or at some later point in time).

FIGURE 17.2

Two sample transactions. (a) Transaction T_1. (b) Transaction T_2.

(a) T_1

```
read_item (X);
X:=X-N;
write_item (X);
read_item (Y);
Y:=Y+N;
write_item (Y);
```

(b) T_2

```
read_item (X);
X:=X+M;
write_item (X);
```

Introduction to Transaction Processing (7)

Why Concurrency Control is needed:

- **The Lost Update Problem.**

  This occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.

- **The Temporary Update (or Dirty Read) Problem.**

  This occurs when one transaction updates a database item and then the transaction fails for some reason (see Section 17.1.4). The updated item is accessed by another transaction before it is changed back to its original value.

Introduction to Transaction Processing (8)

Why Concurrency Control is needed (cont.):

- **The Incorrect Summary Problem .**

  If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated.

FIGURE 17.3

Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem.

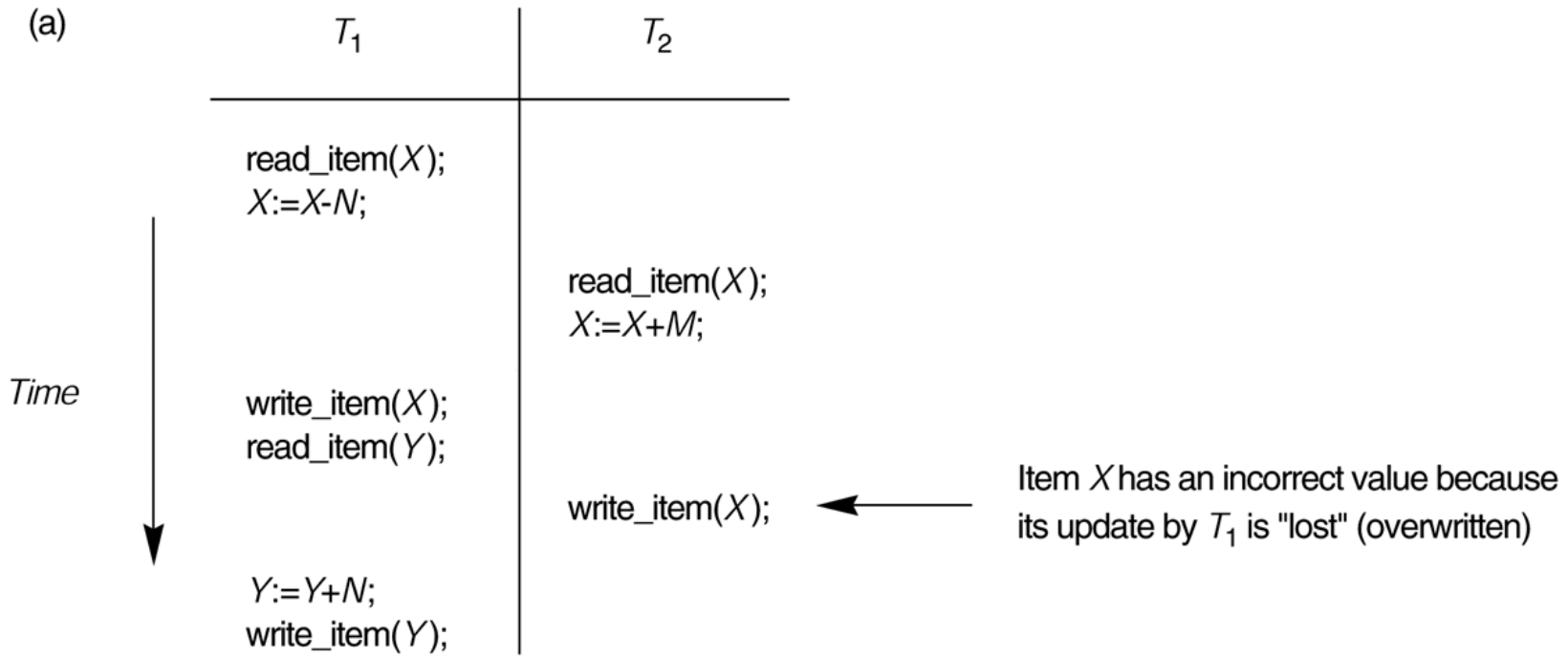


FIGURE 17.3 (continued)

Some problems that occur when concurrent execution is uncontrolled. (b) The temporary update problem.

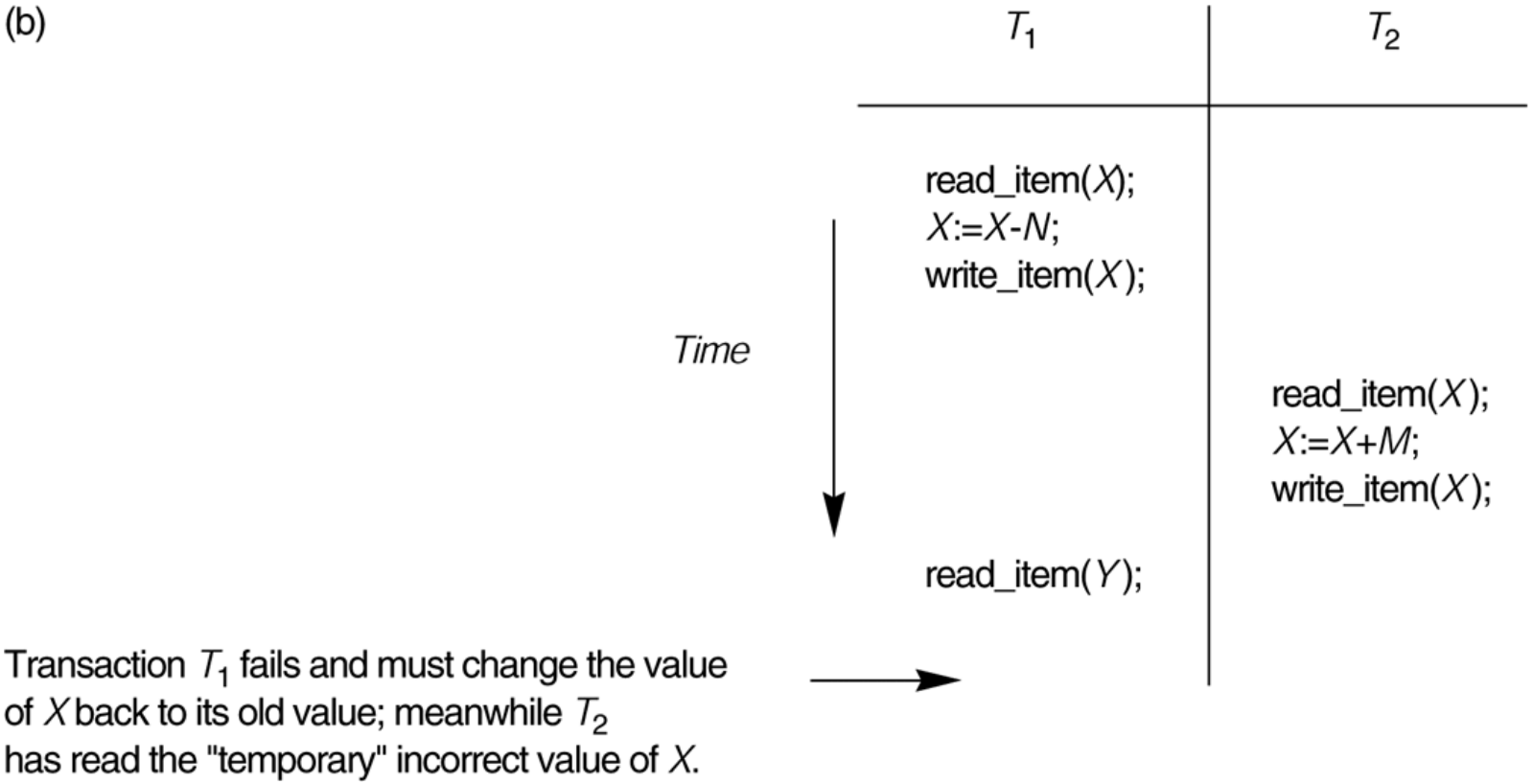
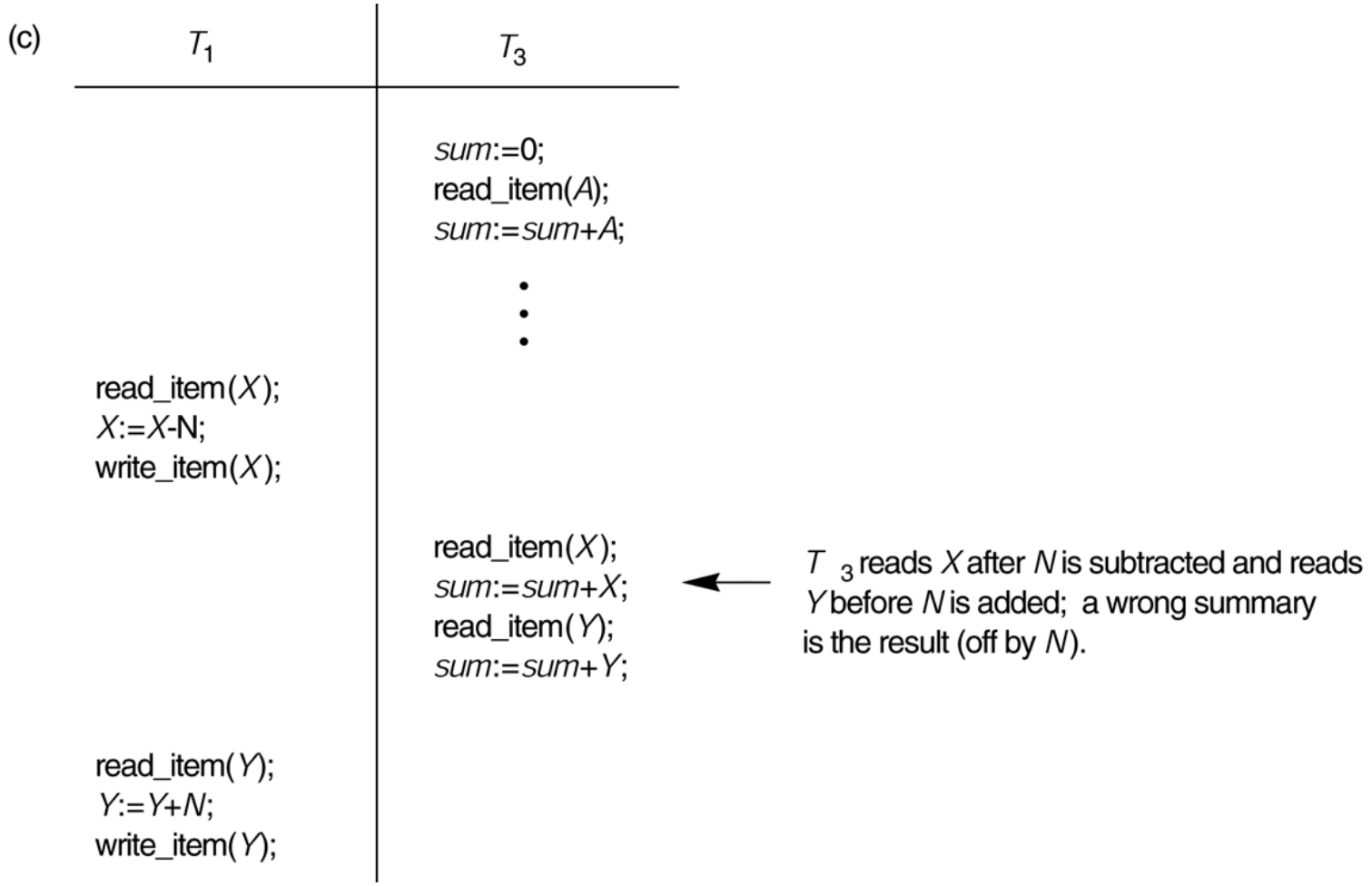


FIGURE 17.3 (continued)

Some problems that occur when concurrent execution is uncontrolled. (c) The incorrect summary problem.



Introduction to Transaction Processing (11)

Why recovery is needed:

(What causes a Transaction to fail)

1. **A computer failure (system crash):** A hardware or software error occurs in the computer system during transaction execution. If the hardware crashes, the contents of the computer's internal memory may be lost.
2. **A transaction or system error :** Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. In addition, the user may interrupt the transaction during its execution.

Introduction to Transaction Processing (12)

Why recovery is needed (cont.):

3. **Local errors or exception conditions** detected by the transaction:

   - certain conditions necessitate cancellation of the transaction. For example, data for the transaction may not be found. A condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal from that account, to be canceled.

   - a programmed abort in the transaction causes it to fail.

4. **Concurrency control enforcement:** The concurrency control method may decide to abort the transaction, to be restarted later, because it violates serializability or because several transactions are in a state of deadlock (see Chapter 18).

Introduction to Transaction Processing (13)

Why recovery is needed (cont.):

5. **Disk failure:** Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.
6. **Physical problems and catastrophes:** This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

2 Transaction and System Concepts (1)

A **transaction** is an atomic unit of work that is either completed in its entirety or not done at all. For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts.

Transaction states:

- Active state
- Partially committed state
- Committed state
- Failed state
- Terminated State

Transaction and System Concepts (2)

Recovery manager keeps track of the following operations:

- **begin_transaction:** This marks the beginning of transaction execution.
- **read or write:** These specify read or write operations on the database items that are executed as part of a transaction.
- **end_transaction:** This specifies that read and write transaction operations have ended introduced by the transaction can and marks the end limit of transaction execution. At this point it may be necessary to check whether the changes be permanently applied to the database or whether the transaction has to be aborted because it violates concurrency control or for some other reason.

Transaction and System Concepts (3)

Recovery manager keeps track of the following operations (cont):

- **commit_transaction:** This signals a *successful end* of the transaction so that any changes (updates) executed by the transaction can be safely **committed** to the database and will not be undone.
- **rollback (or abort):** This signals that the transaction has *ended unsuccessfully,* so that any changes or effects that the transaction may have applied to the database must be *undone.*

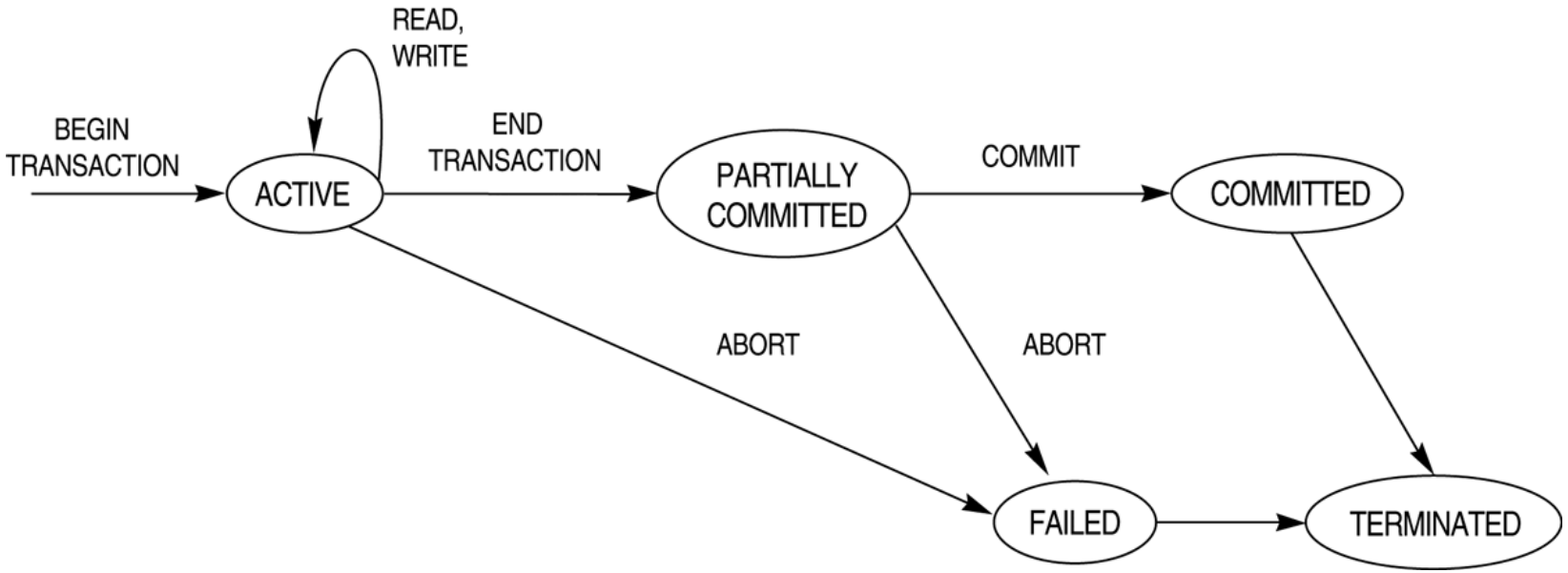
Transaction and System Concepts (4)

Recovery techniques use the following operators:

- **undo:** Similar to rollback except that it applies to a single operation rather than to a whole transaction.
- **redo:** This specifies that certain *transaction operations* must be *redone* to ensure that all the operations of a committed transaction have been applied successfully to the database.

FIGURE 17.4

State transition diagram illustrating the states for transaction execution.



Transaction and System Concepts (6)

The System Log

- **Log or Journal** : The log keeps track of all transaction operations that affect the values of database items. This information may be needed to permit recovery from transaction failures. The log is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure. In addition, the log is periodically backed up to archival storage (tape) to guard against such catastrophic failures.
- T in the following discussion refers to a unique **transaction-id** that is generated automatically by the system and is used to identify each transaction:

Transaction and System Concepts (7)

The System Log (cont):

Types of log record:

1. [start_transaction,T]: Records that transaction T has started execution.
2. [write_item,T,X,old_value,new_value]: Records that transaction T has changed the value of database item X from old_value to new_value.
3. [read_item,T,X]: Records that transaction T has read the value of database item X.
4. [commit,T]: Records that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
5. [abort,T]: Records that transaction T has been aborted.

Transaction and System Concepts (8)

The System Log (cont):

- protocols for recovery that avoid cascading rollbacks do not require that read operations be written to the system log, whereas other protocols require these entries for recovery.
- strict protocols require simpler write entries that do not include new_value (see Section 17.4).

Transaction and System Concepts (9)

Recovery using log records:

If the system crashes, we can recover to a consistent database state by examining the log and using one of the techniques described in Chapter 19.

1. Because the log contains a record of every write operation that changes the value of some database item, it is possible to **undo** the effect of these write operations of a transaction T by tracing backward through the log and resetting all items changed by a write operation of T to their old_values.
2. We can also **redo** the effect of the write operations of a transaction T by tracing forward through the log and setting all items changed by a write operation of T (that did not get done permanently) to their new_values.

Transaction and System Concepts (10)

Commit Point of a Transaction:

- **Definition:** A transaction T reaches its **commit point** when all its operations that access the database have been executed successfully *and* the effect of all the transaction operations on the database has been recorded in the log. Beyond the commit point, the transaction is said to be **committed,** and its effect is assumed to be *permanently recorded* in the database. The transaction then writes an entry [commit,T] into the log.
- **Roll Back of transactions:** Needed for transactions that have a [start_transaction,T] entry into the log but no commit entry [commit,T] into the log.

Transaction and System Concepts (11)

Commit Point of a Transaction (cont):

- **Redoing transactions:** Transactions that have written their commit entry in the log must also have recorded all their write operations in the log; otherwise they would not be committed, so their effect on the database can be *redone* from the log entries. (Notice that the log file must be kept on disk. At the time of a system crash, only the log entries that have been *written back to disk* are considered in the recovery process because the contents of main memory may be lost.)
- **Force writing a log:** *before* a transaction reaches its commit point, any portion of the log that has not been written to the disk yet must now be written to the disk. This process is called force-writing the log file before committing a transaction.

3 Desirable Properties of Transactions (1)

ACID properties:

- **Atomicity**: A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.

- **Consistency preservation**: A correct execution of the transaction must take the database from one consistent state to another.

Desirable Properties of Transactions (2)

ACID properties (cont.):

- **Isolation**: A transaction should not make its updates visible to other transactions until it is committed; this property, when enforced strictly, solves the temporary update problem and makes cascading rollbacks of transactions unnecessary (see Chapter 21).
- **Durability or permanency**: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

Example of Fund Transfer

- 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*)
- **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**
- **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.

Example of Fund Transfer (Cont.)

- 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*)
- **Consistency requirement** in above example:
  - the **sum of A and B is unchanged** by the execution of the transaction
- In general, consistency requirements include
  - **Explicitly** specified **integrity constraints** such as primary keys and foreign keys
  - **Implicit** integrity constraints
    - e.g. **sum of balances of all accounts**, minus sum of loan amounts **must equal value of cash-in-hand**
  - A transaction 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

Example of Fund Transfer (Cont.)

- **Isolation requirement** — if between steps 3 and 6, 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	**T2**
1.	**read**(A)	
2.	$A := A - 50$	
3.	**write**(A)	
		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**, as we will see later.

Concurrent Executions

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

Schedule

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

- A **schedule** (or **history**) S of n transactions T1, T2, ..., Tn :

  It is an ordering of the operations of the transactions subject to the constraint that, for each transaction Ti that participates in S, the operations of T1 in S must appear in the same order in which they occur in T1. Note, however, that operations from other transactions Tj can be interleaved with the operations of Ti in S.

5 Characterizing Schedules based on Serializability (1)

- **Serial schedule**: A schedule S is **serial** if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule. Otherwise, the schedule is called **nonserial schedule.**
- **Serializable schedule**: A schedule S is **serializable** if it is equivalent to some serial schedule of the same n transactions.

Schedule 1

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

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

A	B	A+B	TRANSACTION	REMARK
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

@Silberchatz, Korth, Sudarshan and NPTEL

Schedule 2

- A serial schedule where T_2 is followed by T_1

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

A	B	A+B	TRANSACTION	REMARK
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

@Silberchatz, Korth, Sudarshan and NPTEL

Schedule 3

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

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

A	B	A+B	TRANSACTION	REMARK
100	200	300	@start	
50	200	250	T1, write A	
45	200	245	T2, write A	
45	250	295	T1, write B	@commit
45	255	300	T2, write B	@commit

Consistent @ Commit

Inconsistent @ Transit

Inconsistent @ Commit

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

Schedule 4

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

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

A	B	A+B	TRANSACTION	REMARK
100	200	300	@start	
90	200	290	T2, write A	
50	200	250	T1, write A	
50	250	300	T1, write B	@commit
50	260	310	T2, write B	@commit

Consistent @ Commit

Inconsistent @ Transit

Inconsistent @ Commit

@Silberchatz, Korth, Sudarshan and 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**

Simplified view of transactions

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

Conflicting Instructions

- Instructions l_i and l_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both l_i and l_j, and at least one of these instructions wrote Q.
  1. l_i = **read**(Q), l_j = **read**(Q). l_i and l_j **don't conflict**.
  2. l_i = **read**(Q), l_j = **write**(Q). They **conflict**.
  3. l_i = **write**(Q), l_j = **read**(Q). They **conflict**
  4. l_i = **write**(Q), l_j = **write**(Q). They **conflict**
- Intuitively, a conflict between l_i and l_j forces a (logical) temporal order between them.
  - If l_i and l_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.

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**

Conflict Serializability (Cont..)

- Schedule 3 can be transformed into Schedule 6 –a serial schedule where *T*2follows *T*1, by a series of swaps of non-conflicting instructions.
- Swap T1.read(B) and T2.write(A)
- Swap T1.read(B) and T2.read(A)
- Swap T1.write(B) and T2.write(A)
- Swap T1.write(B) and T2.read(A)

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

Schedule 3

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

Schedule 6

Conflict Serializability (Cont..)

- Example of a schedule that is not conflict serializable:

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

We are unable to swap instructions in the above schedule to obtain either the serial schedule <*T*3, *T*4>, or the serial schedule < *T*4, *T*3>

Serializability

- Are all serializable schedules conflict-serializable? No.
- Consider the following schedule for a set of three transactions.

  $w1(A),w2(A),w2(B),w1(B),w3(B)$
- 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 *transaction* operations, the schedule has the same outcome as the serial schedule:

  $w1(A),w1(B),w2(A),w2(B),w3(B)$

Testing for Conflict Serializability

- Consider a schedule for a set of transactions $T_1, T_2, ..., T_n$

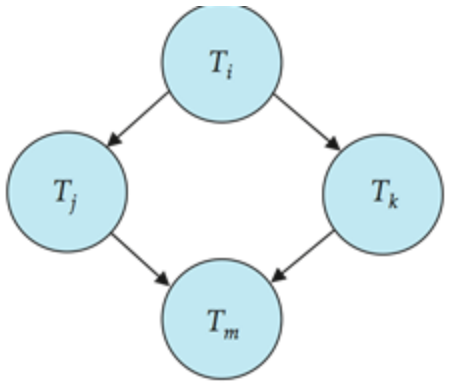
- A *precedence graph* for the schedule is a directed graph which has:
  - A vertex for each transaction
  - An edge from T_i to T_j if they contain conflicting instructions, and the conflicting instruction from T_i accessed the data item before the conflicting instruction from T_j.
    - If executes T_i **write**(Q) before T_j executes **read**(Q)
    - If executes T_i **read**(Q) before T_j executes **write**(Q)
    - If executes T_i **write**(Q) before T_j executes **write**(Q).

- We may label the arc by the item that was accessed.

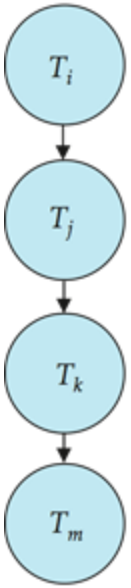
- Duplicate edges may results from the above, but can be deleted.

Testing for Conflict Serializability

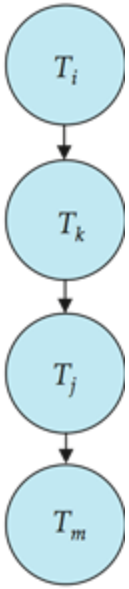
- A schedule is conflict serializable if and only if its precedence graph is acyclic
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph. For example, a serializability order for the schedule (a) would be one of either (b) or (c)



(a)



(b)



(c)

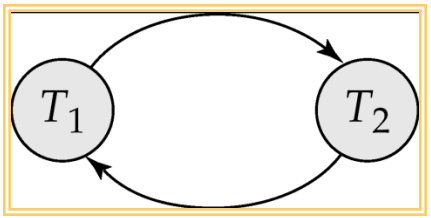
Testing for Conflict Serializability

- Build a directed graph, with a vertex for each transaction.
- Go through each operation of the schedule. If the operation is of the form *wi*(*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 *rj*(*X*) or *wj*(*X*). For each such subsequent operation, add a directed edge in the graph from *Ti* to *Tj*.
- If the operation is of the form *ri*(*X*), find each subsequent *write* to the same data element *X* by a different transaction: that is, anything of the form *wj*(*X*). For each such subsequent write, add a directed edge in the graph from *Ti* to *Tj*.
- 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

- Example:

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)	
	$B := B + temp$
	write(B)

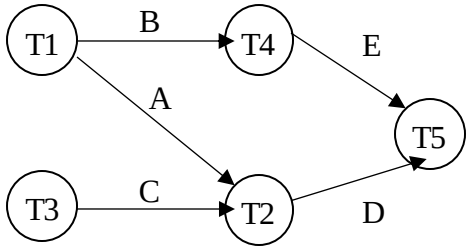


Testing for Conflict Serializability

- Consider the following schedule:

$w1(A),r2(A),w1(B),w3(C),r2(C),r4(B),w2(D),w4(E),r5(D),w5(E)$

- We start with an empty graph with five vertices labeled $T1,T2,T3,T4,T5$.



T1	T2	T3	T4	T5
Write(A)				
	Read(A)			
Write(B)				
		Write(C)		
	Read(C)			
			Read(B)	
	Write(D)			
			Write(E)	
				Read(D)
				Write(E)

- We go through each operation in the schedule:
- $w1(A)$: A is subsequently read by $T2$, so add edge $T1 \rightarrow T2$
- $r2(A)$: no subsequent writes to A, so no new edges
- $w1(B)$: B is subsequently read by $T4$, so add edge $T1 \rightarrow T4$
- $w3(C)$: C is subsequently read by $T2$, so add edge $T3 \rightarrow T2$
- $r2(C)$: no subsequent writes to C, so no new edges
- $r4(B)$: no subsequent writes to B, so no new edges
- $w2(D)$: D is subsequently read by $T5$, so add edge $T2 \rightarrow T5$
- $w4(E)$: E is subsequently written by $T5$, so add edge $T4 \rightarrow T5$
- $r5(D)$: no subsequent writes to D, so no new edges
- $w5(E)$: no subsequent operations on E, so no new edges

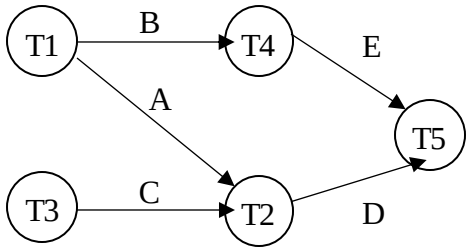
Source: http://www.cburch.com/cs/340/reading/serial/

Testing for Conflict Serializability (Cont)

- Consider the following schedule:

*w*1(*A*),*r*2(*A*),*w*1(*B*),*w*3(*C*),*r*2(*C*),*r*4(*B*),*w*2(*D*),*w*4(*E*),*r*5(*D*),*w*5(*E*)

- We start with an empty graph with five vertices labeled *T*1,*T*2,*T*3,*T*4,*T*5.



- 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/

Exercise

- **Consider the following schedules involving two transactions. Which one of the following statement is true?**

S1: $R_1(X)$ $R_1(Y)$ $R_2(X)$ $R_2(Y)$ $W_2(Y)$ $W_1(X)$
S2: $R_1(X)$ $R_2(X)$ $R_2(Y)$ $W_2(Y)$ $R_1(Y)$ $W_1(X)$

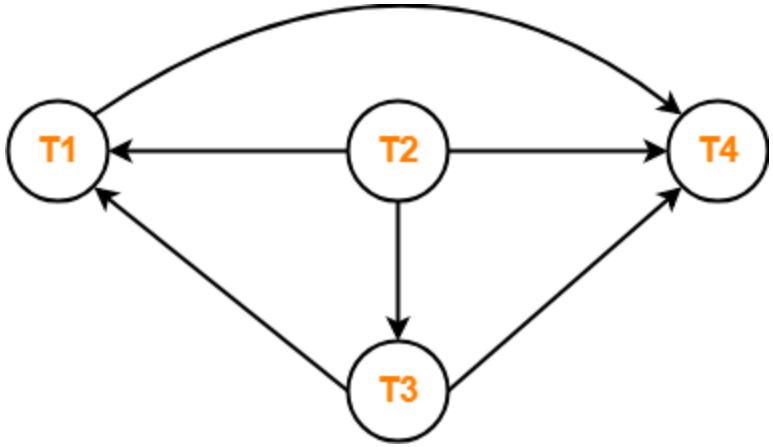
1) Both S1 and S2 are conflict serializable
2) Only S1 is conflict serializable
3) Only S2 is conflict serializable

Ans- 3

Exercise

Check whether the given schedule S is conflict serializable and recoverable or not-

T1	T2	T3	T4
	R(X)		
		W(X)	
		Commit	
W(X)			
Commit			
	W(Y)		
	R(Z)		
	Commit		
			R(X)
			R(Y)
			Commit



- Clearly, there exists no cycle in the precedence graph.
- Therefore, the given schedule S is conflict serializable.

Characterizing Schedules based on Serializability

- Being serializable is not the same as being serial
- Serializability helps to ensure Isolation and Consistency of a schedule
- Yet, the Atomicity and Consistency may be compromised in the face of system failures
- Being serializable implies that the schedule is a correct schedule.
  - It will leave the database in a consistent state.
  - The interleaving is appropriate and will result in a state as if the transactions were serially executed, yet will achieve efficiency due to concurrent execution.

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 schedule

- One where no transaction needs to be rolled back.

  A schedule S is **recoverable** if no transaction T in S commits until all transactions T' that have written an item that T reads have committed.

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

- The following schedule is not recoverable if *T9* commits immediately after the read(A) operation

T_8	T_9
read(A)	
write(A)	
	read(A)
	commit
read(B)	

- 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 Rollback

- A schedule in which uncommitted transactions that read an item from a failed transaction must be rolled back.
- 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)		
	read(A)	
	write(A)	
		read(A)

If T_{10} fails, T_{11} and T_{12} must also be rolled back.

This can lead to the undoing of a significant amount of work

Cascadeless Rollback

- for each pair of transactions *Ti* and *Tj* such that *Tj* reads a data item previously written by *Ti*, the commit operation of *Ti* appears before the read operation of *Tj*.
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

Characterizing Schedules based on Recoverability (3)

- **Strict Schedules:** A schedule in which a transaction can neither read or write an item X until the last transaction that wrote X has committed.

View Serializability

- **View equivalence**: A less restrictive definition of equivalence of schedules

- **View serializability:** definition of serializability based on view equivalence. A schedule is *view serializable* if it is *view equivalent* to a serial schedule.

View Serializability

- Let S and S' be two schedules with the same set of transactions.

- S and S' are said to be *view equivalent* if the following three conditions are met:
  1. For each data item Q, if transaction T_i reads the initial value of Q in schedule S, then transaction T_i reads the initial value of Q in schedule S'.
  2. For each data item Q, the transaction (if any) that performs the final **write**(Q) operation in schedule S performs the final **write**(Q) operation in schedule S'.
  3. For each data item Q, if transaction T_i executes **read**(Q) in schedule S, and that value was produced by transaction T_j in S, then transaction T_i reads the value of Q that was produced by transaction T_j in schedule S'.

- View equivalence is also based only on reads and writes, but it considers details of those reads and writes a bit more closely:
  - It is therefore less conservative, i.e., fewer false negatives.
- Every conflict-serializable schedule is also view-serializable
- However, a schedule may be view-serializable, but not conflict-serializable

Characterizing Schedules based on Serializability (8)

The premise behind view equivalence:

- As long as each read operation of a transaction reads the result of *the same write operation* in both schedules, the write operations of each transaction musr produce the same results.
- **"The view"**: the read operations are said to see the *the same view* in both schedules.

View Serializability

- **Thumb Rules**

1. "Initial readers must be same for all the data items"
2. "Write-read sequence must be same.".
3. "Final writers must be same for all the data items".
4. All conflict serializable schedules are view serializable.
5. All view serializable schedules may or may not be conflict serializable.
6. If there does not exist any blind write, then the schedule is surely not view serializable. Stop and report your answer.
7. If blind write is present, build a dependency graph. If the graph is acyclic, the schedule is view serailizable.

View Serializability (Cont.)

- Example #1:

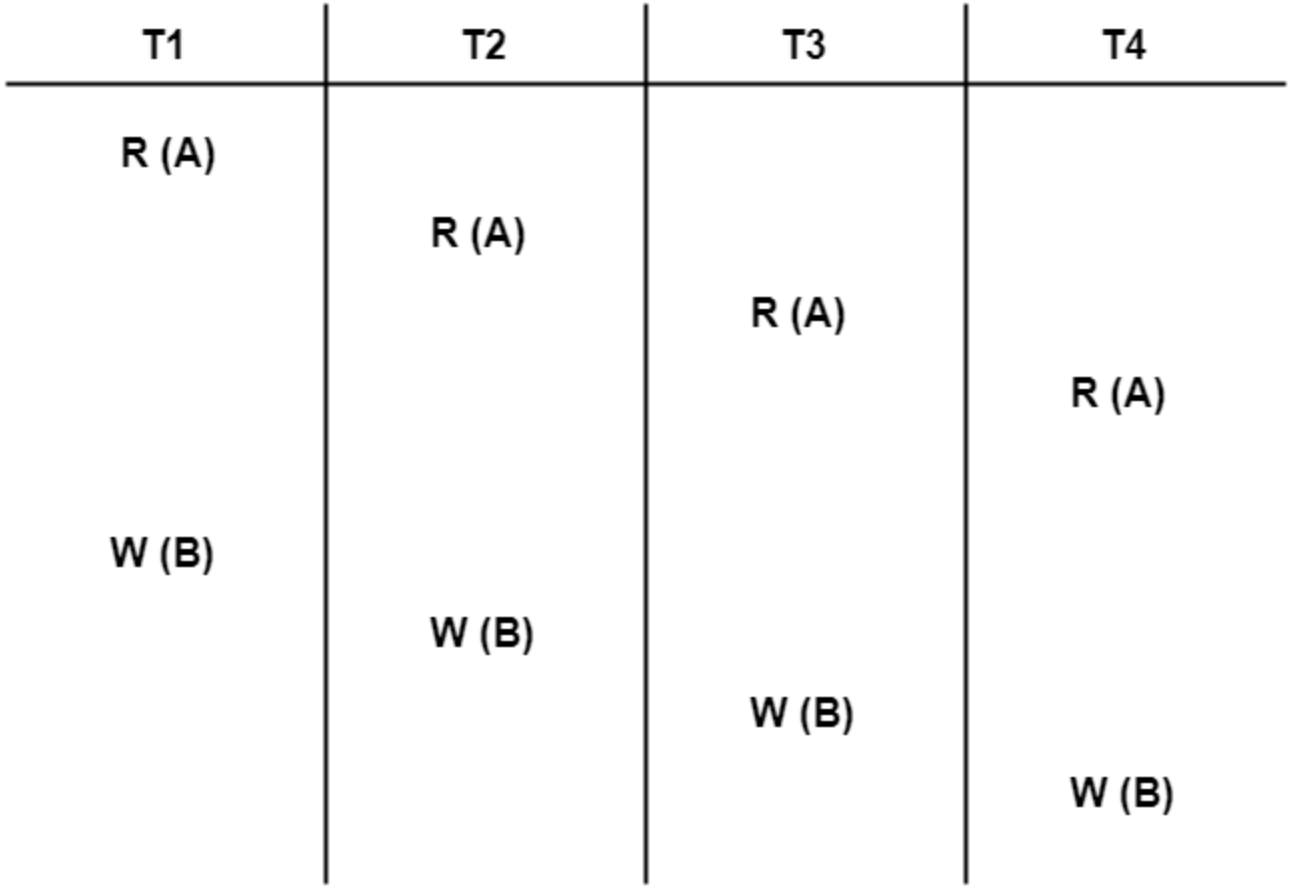
T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

- What serial schedule is the above view-equivalent to?
  - T3–T4–T6
  - The one read(Q) instruction reads the initial value of Q in both schedules and
  - T6 performs the final write of Q in both schedules

- T28 and T29 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**
- Is the above schedule conflict-equivalent to that serial schedule?

View Serializability-Example(cont)

Check whether the given schedule S is conflict serializable and view serializable or not



View Serializability-Example(cont)

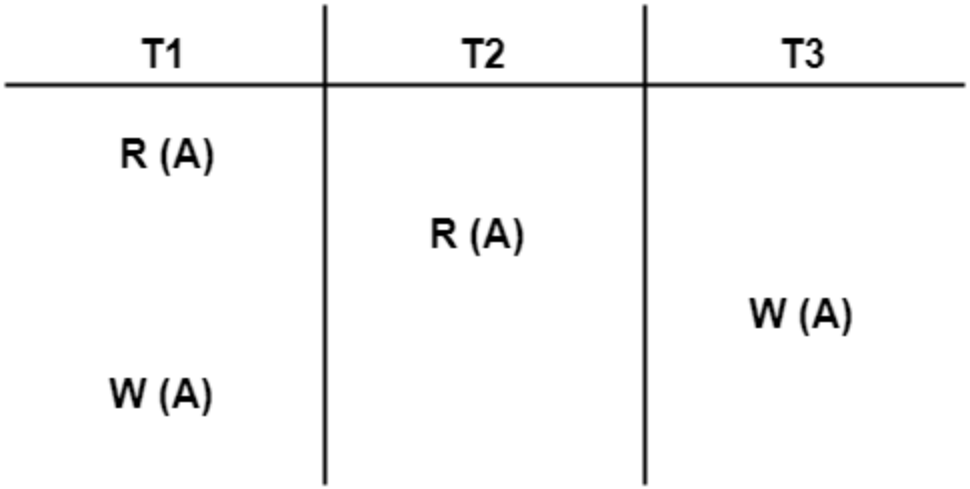
- Solution-

T1	T2	T3	T4
R (A)			
	R (A)		
		R (A)	
			R (A)
W (B)			
	W (B)		
		W (B)	
			W (B)

- Step 1- Check whether schedule is conflict serializable.
- Given Schedule is conflict serializable. Therefore, it is also view serializable.

View Serializability-Example(cont)

Check whether the given schedule S is view serializable or not-



Solution-

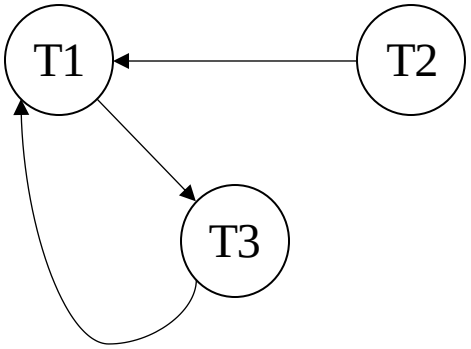
1. Check whether the schedule is conflict serializable.

   Schedule is not conflict serializable since precedence graph contains a cycle .

2. Check for blind writes.

   Transaction T3 contains blind write. Therefore, schedule may or may not be view serializable

View Serializability-Example(cont)



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

Solution-

3. Check for view serializablity using dependency graph.

1. Draw vertices for each transaction
2. T1 firstly reads A and T3 firstly updates A. So, T1 must execute before T3. Thus, we get the dependency **T1 → T3**
3. Final updation on A is made by the transaction T1. So, T1 must execute after all other transactions. Thus, we get the dependency **(T2, T3) → T1**.
4. There exists no write-read sequence.

There exists a cycle in the dependency graph. Therefore, the given schedule S is not view serializable.

Exercise

Check whether the given schedule S is view serializable or not-

T1	T2
R (A)	
A = A + 10	
	R (A)
	A = A + 10
W (A)	
	W (A)
R (B)	
B = B + 20	
	R (B)
	B = B x 1.1
W (B)	
	W (B)

- Solution-

Step 1-The schedule is not conflict serializable. Therefore, it may or may not be view serializable.

step 2- No Blind Write is present. Therefore, schedule is not view serializable.

Exercise

Check whether the given schedule S is view serializable or not. If yes, then give the serial schedule.

$$S : R_1(A) , W_2(A) , R_3(A) , W_1(A) , W_3(A)$$

Solution-

1- Check whether the schedule is conflict Serializable.

No conflict serializable

2- Blind writes are present in T2.

3. Drawing dependency graph.

T1 firstly reads A and T2 firstly updates A.

So, T1 must execute before T2.

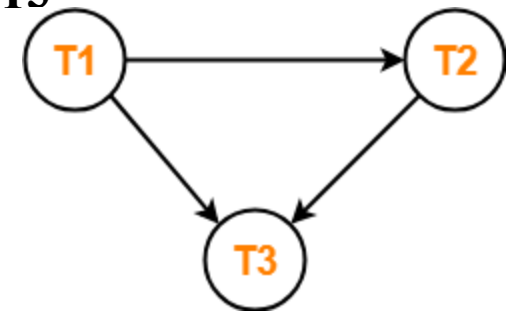
Thus, we get the dependency **T1 → T2**.

So, T3 must execute after all other transactions.

Thus, we get the dependency **(T1, T2) → T3**.

From write-read sequence, we get the dependency **T2 → T3**

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



4. There exists no cycle in the dependency graph.

Therefore, the given schedule S is view serializable.

5. **The serialization order T1 → T2 → T3.**

Characterizing Schedules based on Serializability (9)

Relationship between view and conflict equivalence:

- The two are same under **constrained write assumption** which assumes that if T writes X, it is constrained by the value of X it read; i.e., new X = f(old X)
- Conflict serializability is **stricter** than view serializability. With unconstrained write (or blind write), a schedule that is view serializable is not necessarily conflict serialiable.
- Any conflict serializable schedule is also view serializable, but not vice versa.