CS3402 Chapter 10:
Transaction Management

Single-User versus Multiuser Systems

- A database system is single-user if at most one user at a time can use the system, and it is multiuser if many users can use the system—and hence access the database—concurrently.
- Single-user database system are mostly restricted to personal computer systems; most other database systems are multiuser, for example, an airline reservations system is used by hundreds of users and travel agents concurrently.
- In multiuser systems, hundreds or thousands of users are typically operating on the database by submitting transactions concurrently to the system. So we need transaction concurrency control to make the database system consistent.

What is a Transaction?

- A transaction is an executing program that forms a logical unit of database processing.

  E.g. use of ATM and buy online tickets

- A transaction includes one or more database access operations—these can include insertion, deletion, modification (update), or retrieval operations.

- To specify the transaction boundaries, we use begin transaction and end transaction statements in an application program; in this case, all database access operations between the two are considered as forming one transaction.

What is a transaction?

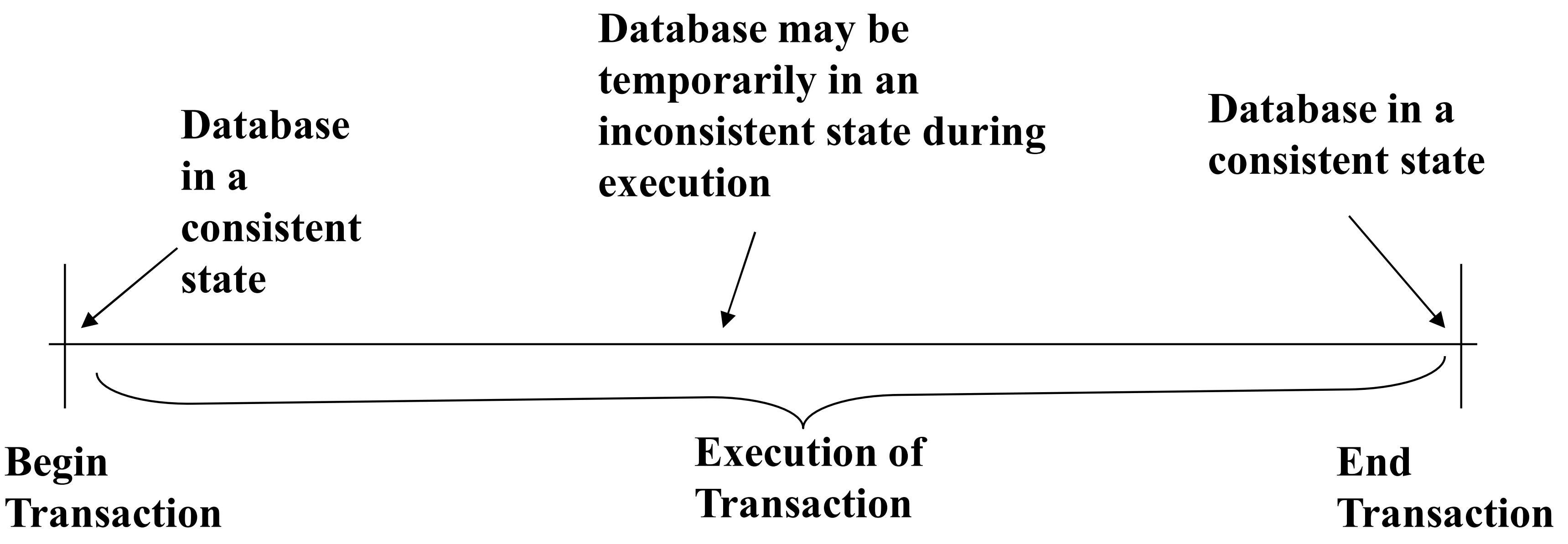
- Structurally, each transaction is a process and consists of atomic steps. Each atomic step is called an operation.
- A transaction = database operations + transaction operations

- Database operations: read and write operations on a database
  - Read operation: to read the value of a data item or a group of items, e.g., SELECT
  - Write operation: to create a new value for a data item or a group of items, e.g., UPDATE
  - In between the read/write operations, there may be computation

- Transaction operations: begin and end transaction (commit or abort)
  - For transaction management (where to start and where to finish)
  - The new values from a transaction will become permanent only if the transaction is committed successfully

Transaction Structure & Database Consistency

Example: T1: Begin; R(a); R(b)； c= a + b; W(c); End


Database in a consistent state
Database may be temporarily in an inconsistent state during execution
Database in a consistent state
Begin Transaction
Execution of Transaction
End Transaction

The whole transaction is considered as an atomic unit

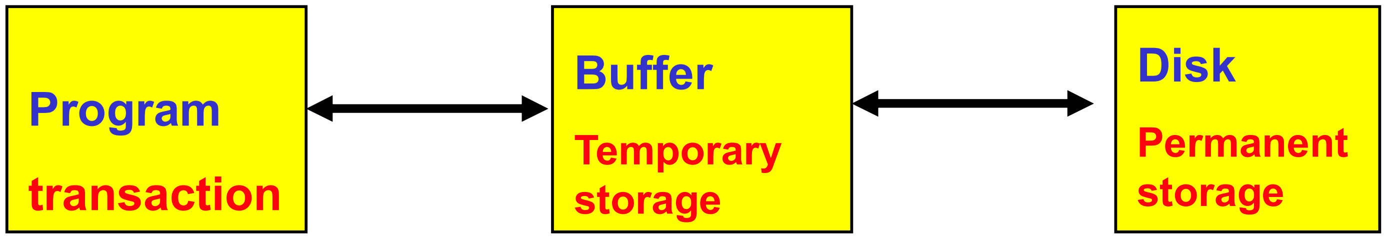
- **Partial results are not allowed and is considered to be incorrect**
- **Atomicity: All or nothing**

Read and Write Operations

- Data are resided on disk and the basic unit of data transferring from the disk to the main memory is one disk block
- In general, a data item (what is read or written) will be the field/fields of some records in the database (in a disk block)
- Read (X)/Read_item (X): command includes the following steps:
  - Find the address of the disk block that contains item X
  - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer)
  - Search for the required value in the buffer
  - Copy item X from the buffer to the program variable named X

Read and Write Operations

- Write(X)/Write_item(X): command includes the following steps:
  - Find the address of the disk block that contains item X
  - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer)
  - Search for the required value in the buffer
  - Copy item X from the program variable named X into its correct location in the buffer
  - Store the updated block from the buffer back to disk (either immediately or at some later point in time)



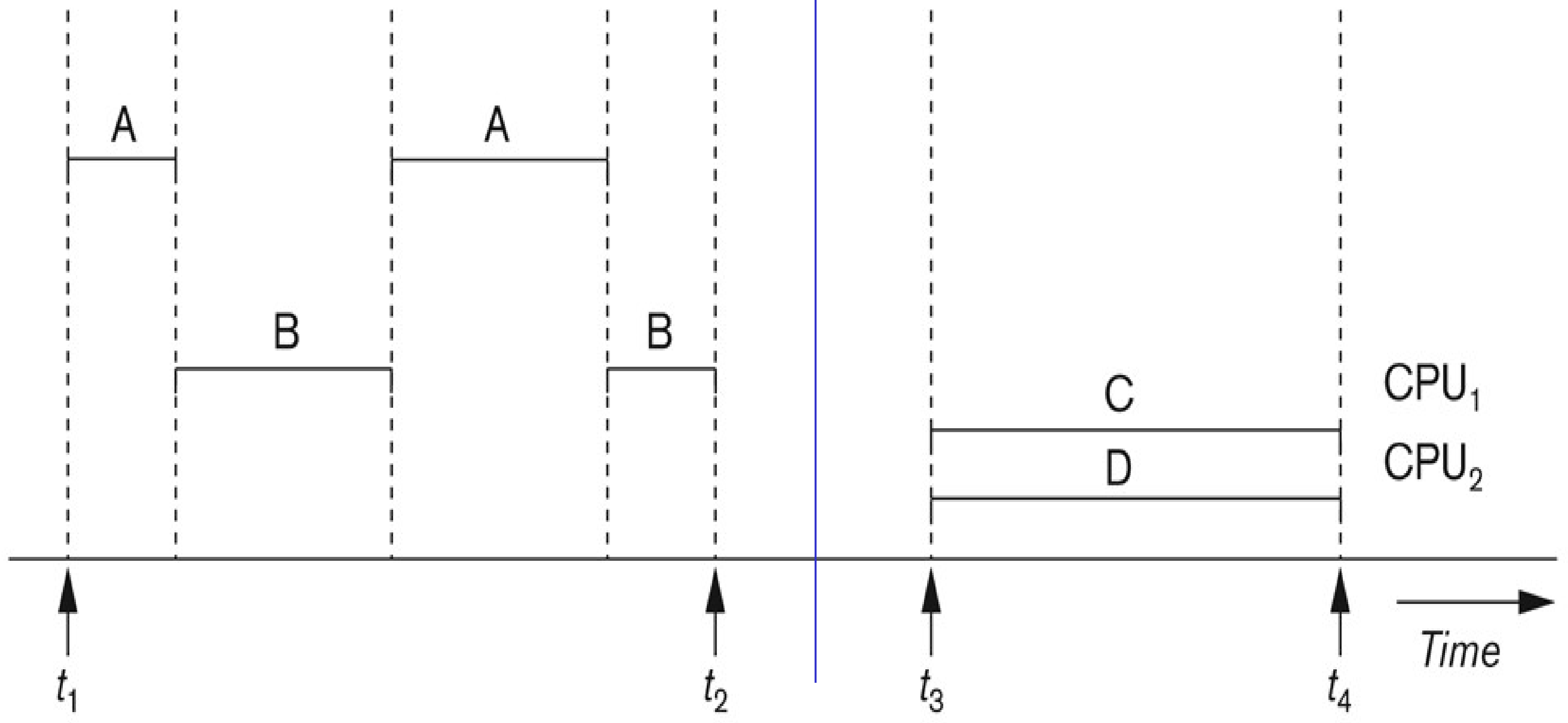
How to improve transaction processing performance?

Serial Schedule Vs. Serializable Schedule

Transaction Processing Performance

- Multiuser systems:
  - Many users (transactions) can access the database concurrently at the same time
- Serial execution: execute transitions one by one (slow)
- Concurrent execution:
  - Interleaved processing:
    - Concurrent execution of processes/transactions are interleaved in a single CPU system
  - Parallel processing:
    - Processes/transactions are concurrently executed in multiple CPUs system
- Higher Concurrency (more than one transaction is executing)
  - Better performance, i.e., lower response time
  - Problem: difficult to maintain database consistency if it is not well scheduled

Interleaved vs Parallel



While waiting disk data, the CPU processes another transaction

Figure 17.1
Interleaved processing versus parallel processing of concurrent transactions.

Transaction Schedule

- Transaction schedule
  - When transactions are executing concurrently in an interleaved fashion or serially, the order of execution of operations from all transactions forms a transaction schedule
- A schedule S of *n* transactions *T1, T2, …, Tn* is an ordering of the operations of the transactions subject to the constraint:
  - For each transaction *Ti* that participates in S, the operations of *Ti* in S must appear in the same order as in *Ti* (operations from other transactions *Tj* can be interleaved with the operations of *Ti* in S)
- A concurrent schedule: a new transaction starts BEFORE the completion of the current transaction
- A serial schedule: a new transaction only starts AFTER the current transaction is completed

Schedule Examples

- T1: Read(x), Write(y), Write(x)
- T2: Read(x), Read(y), Write(y)

T1	T2
Read(x)	
Write(y)	
Write(x)	
	Read(x)
	Read(y)
	Write(y)

T1	T2
Read(x)	
Write(A)	
	Read(x)
Read(B)	
	Read(y)
	Write(y)

Execute time

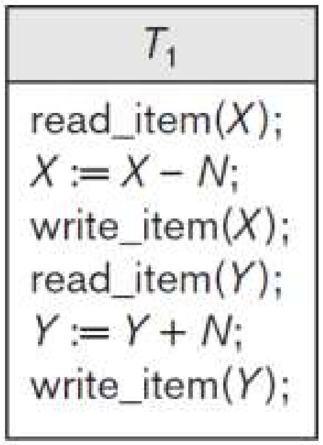
serial schedule

concurrent schedule

Consistency Problems in Concurrent Schedule

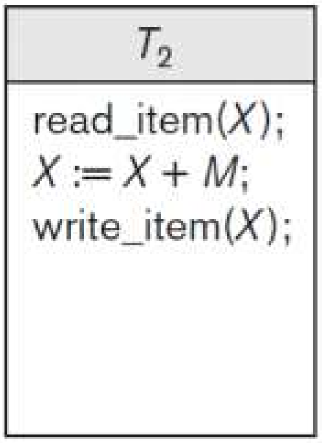
- Several problems can occur when concurrent transactions execute in an uncontrolled manner. We illustrate some of these problems by referring to a much simplified airline reservations database in which a record is stored for each airline flight.
- X, Y are database items to represent the number of reserved seats on each flight.
- Figure (a) shows a transaction T1 that transfers N reservations from flight(X) to flight(Y). Figure (b) shows a simpler transaction T2 that just reserves M seats on flight (X).

(a)



T_1
read_item(X);
$X := X - N$;
write_item(X);
read_item(Y);
$Y := Y + N$;
write_item(Y);

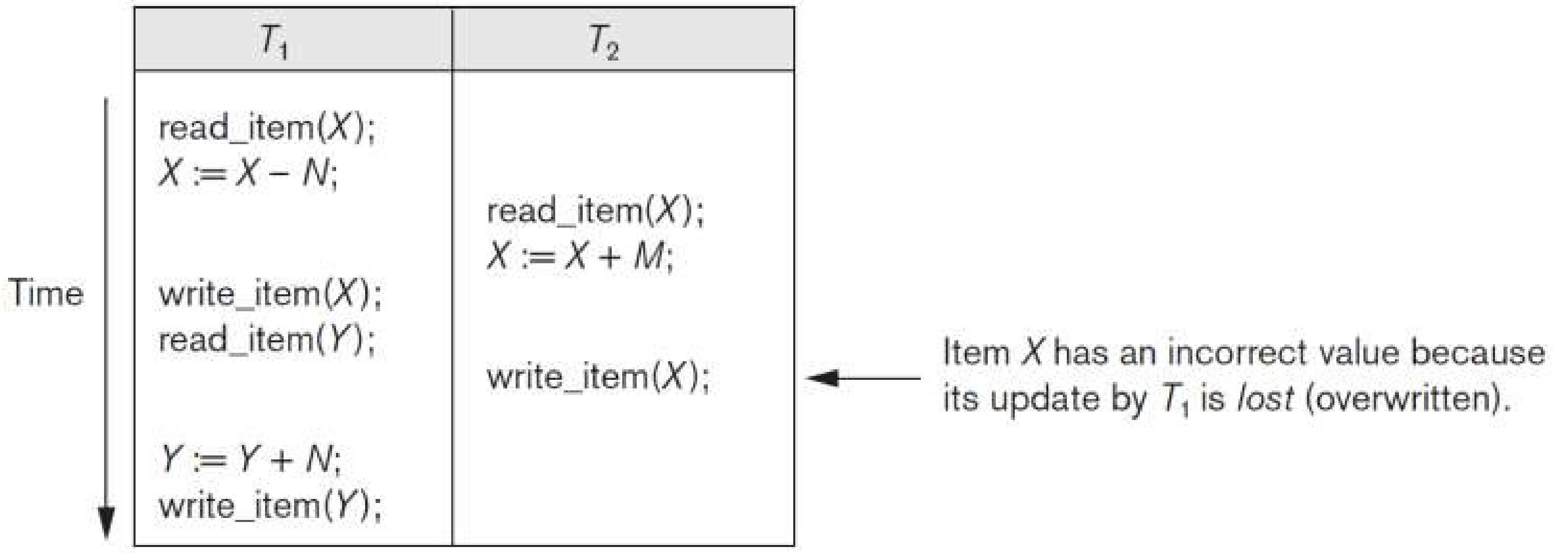
(b)



T_2
read_item(X);
$X := X + M$;
write_item(X);

Consistency Problems in Concurrent Schedule

- Lost Update Problem (write/write conflicts)
  - 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 (inconsistent)
- For example, if X = 80, Y=40 at the start (originally there were 80 reservations on the flight X, 40 on the flight Y), N = 5 (T1 transfers 5 seat reservations from the flight X to the flight Y), and M = 4 (T2 reserves 4 seats on X), the final result should be X = 79, Y=45. However, in the interleaving of operations, it is X = 84 because the update in T1 that removed the five seats from X was lost.



T_1	T_2
read_item(X); $X := X - N$;	
	read_item(X); $X := X + M$;

write_item(X);
read_item(Y);	

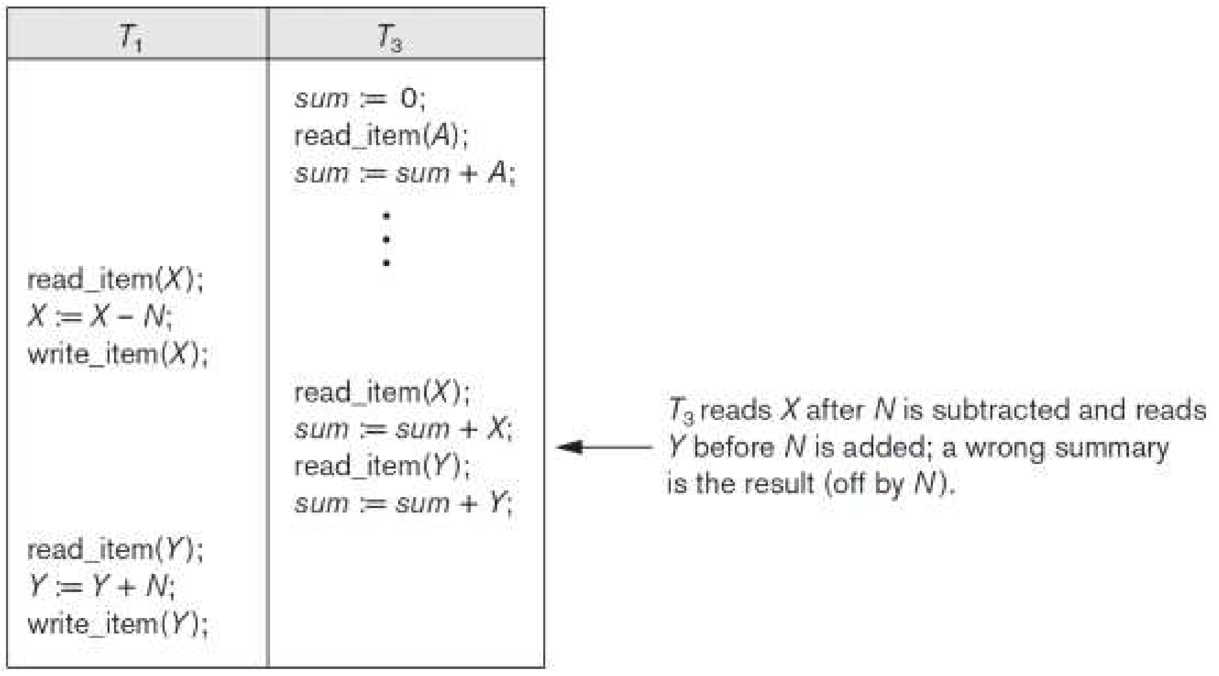
	write_item(X);
$Y := Y + N$; write_item(Y);	

Item X has an incorrect value because its update by T_1 is *lost* (overwritten).

Consistency Problems in Concurrent Schedule

- Incorrect Summary Problem (read/write conflicts)
  - If one transaction is calculating an aggregate summary function on a number of database items while other transactions are updating some of these items, the aggregate function may calculate some values before they are updated and others after they are updated.

If the interleaving of operations occurs, the result of T3 will be off by an amount N because T3 reads the value of X after N seats have been subtracted from it but reads the value of Y before those N seats have been added to it.



T_1	T_3

	sum := 0;
	read_item(*A*);
	sum := *sum* + *A*;
	⋮

read_item(*X*);
X := *X* − *N*;
write_item(*X*);	

	read_item(*X*);
	sum := *sum* + *X*;
	read_item(*Y*);
	sum := *sum* + *Y*;

read_item(*Y*);
Y := *Y* + *N*;
write_item(*Y*);	

T_3 reads *X* after *N* is subtracted and reads *Y* before *N* is added; a wrong summary is the result (off by *N*).

Read and write operation conflict rules

Operations of different transactions		*Conflict*	*Reason*
read	*read*	No	Because the effect of a pair of *read* operations does not depend on the order in which they are executed
read	*write*	Yes	Because the effect of a *read* and a *write* operation depends on the order of their execution
write	*write*	Yes	Because the effect of a pair of *write* operations depends on the order of their execution

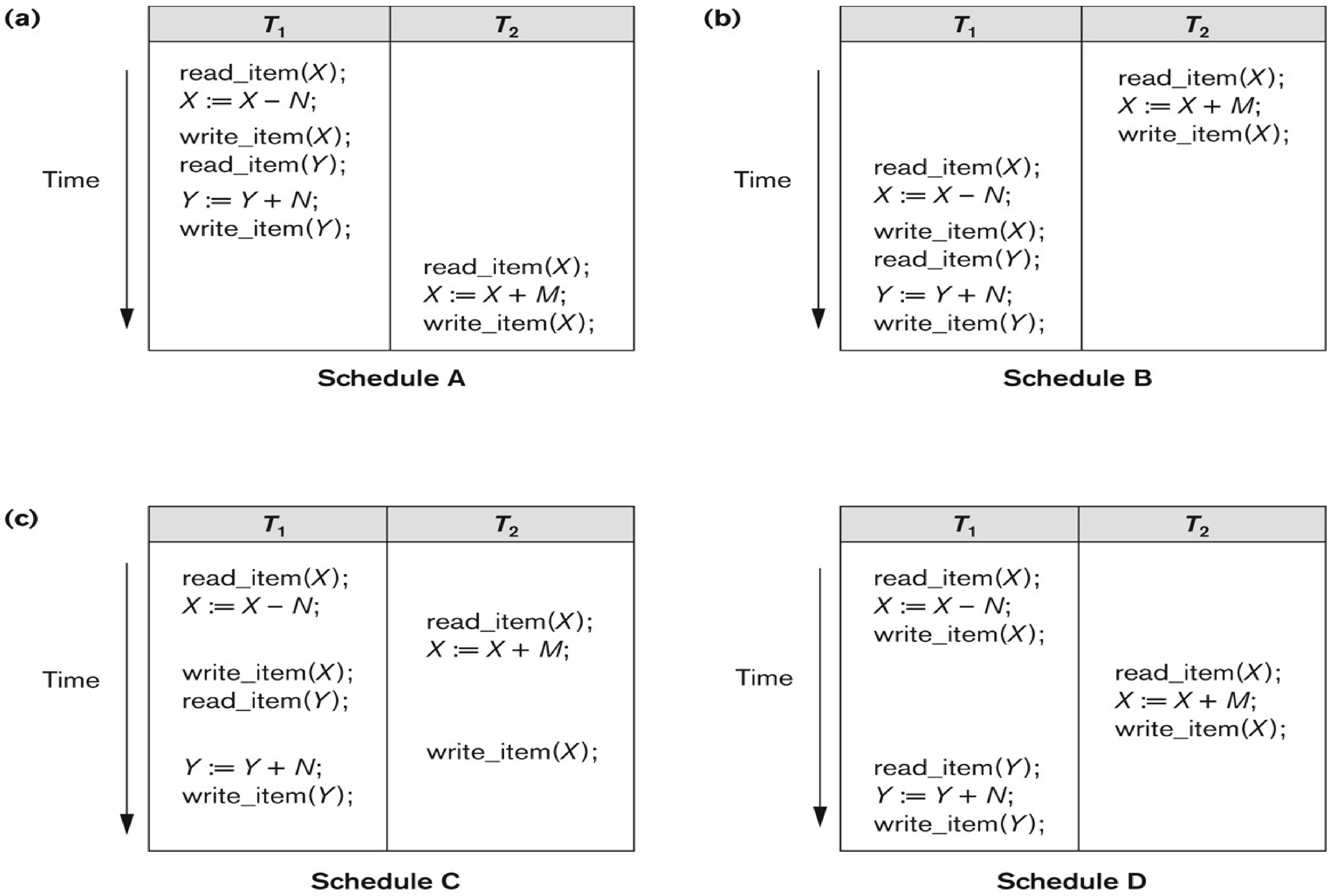
Conflict equivalent:

- **Two schedules are said to be conflict equivalent if the order of any two conflicting operations (RW,WW) is the same in both schedule**

Schedules Classified on Serializability

- 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
  - Serial schedules can maintain the database consistency
    - BUT, poor performance

- Serializable schedule:
  - A concurrent schedule S which is conflict equivalent to a serial schedule.
  - Can guarantee the database consistency and can have better performance.



(a)

T_1	T_2

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

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

Schedule A

(b)

T_1	T_2

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

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

Schedule B

(c)

T_1	T_2
read_item(X); $X := X - N$;	
	read_item(X); $X := X + M$;

write_item(X);
read_item(Y);	

	write_item(X);
$Y := Y + N$; write_item(Y);	

Schedule C

T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);	

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

read_item(Y);
$Y := Y + N$;
write_item(Y);	

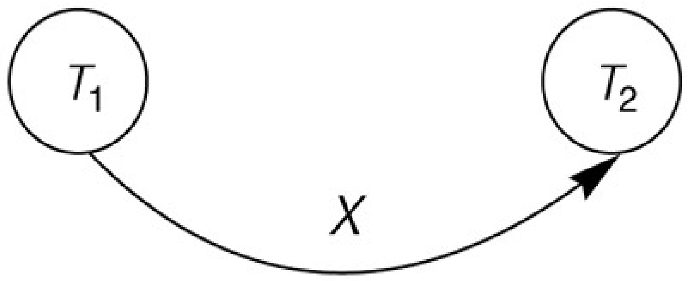
Schedule D

Figure 17.5
Examples of serial and nonserial schedules involving transactions T_1 and T_2. (a) Serial schedule A: T_1 followed by T_2. (b) Serial schedule B: T_2 followed by T_1. (c) Two nonserial schedules C and D with interleaving of operations.

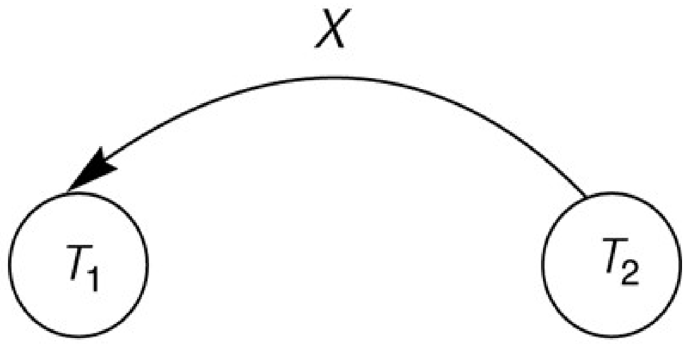
Serialization Graphs

- The determination of a conflict serializable schedule can be done by the use of serialization graph (SG) or called precedence graph
- A serialization graph tells the effective execution order of a set of transactions
- The set of edges consists of all edges $T_i \rightarrow T_j$ for which one of the following three conditions holds:
  - W/R conflict: T_i executes write(x) before T_j executes read(x)
  - R/W conflict: T_i executes read(x) before T_j executes write(x)
  - W/W conflict: T_i executes write(x) before T_j executes write(x)
- Each edge $T_i \rightarrow T_j$ in a SG means that at least one of T_i's operations precede and conflict with one of T_j's operations
- Serializability theorem:
  - A schedule is serializable iff the SG is acyclic

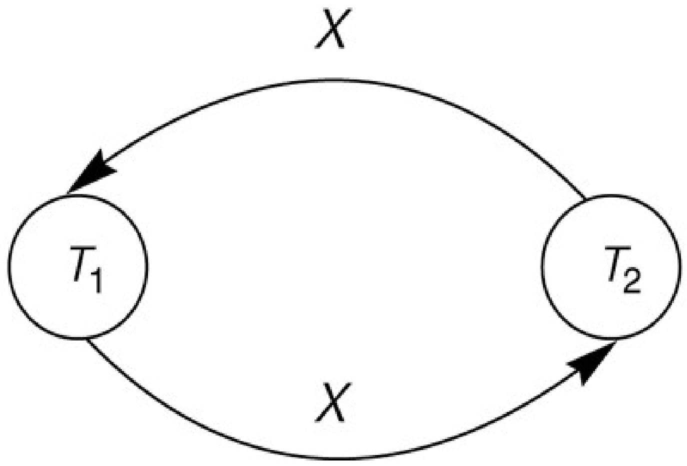
(a)



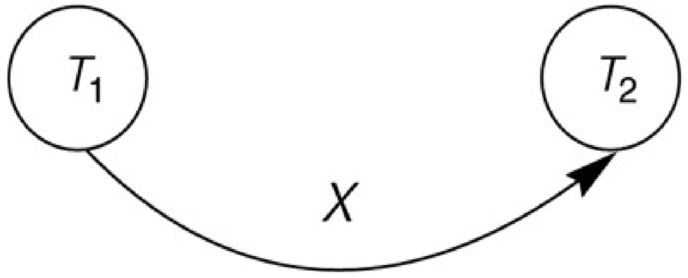
(b)



(c)



(d)



- Constructing the precedence graphs for schedules A and D from Figure 17.5 to test for conflict serializability.
  - (a) Precedence graph for serial schedule A.
  - (b) Precedence graph for serial schedule B.
  - (c) Precedence graph for schedule C (not conflict serializable).
  - (d) Precedence graph for schedule D (conflict serializable, equivalent to schedule A).

Another Example of Serializability Testing

Figure 17.8
Another example of serializability testing. (a) The read and write operations of three transactions T_1, T_2, and T_3. (b) Schedule E. (c) Schedule *F*.

(b)

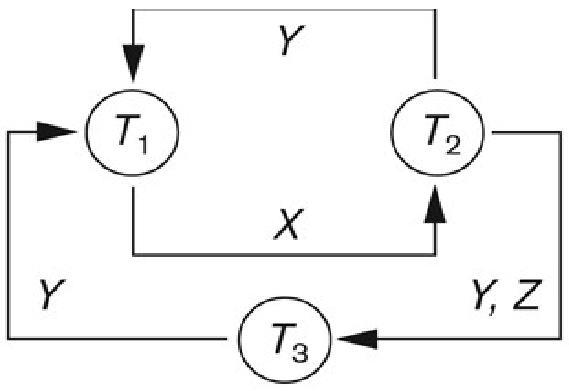
Time ↓

Transaction T_1	Transaction T_2	Transaction T_3
	read_item(*Z*);	
	read_item(*Y*);	
	write_item(*Y*);	
		read_item(*Y*);
		read_item(*Z*);
read_item(*X*);		
write_item(*X*);		
		write_item(*Y*);
		write_item(*Z*);
	read_item(*X*);	
read_item(*Y*);		
write_item(*Y*);		
	write_item(*X*);	

Schedule E

Another Example of Serializability Testing

(d)



Equivalent serial schedules

None

Reason

Cycle $X(T_1 \rightarrow T_2), Y(T_2 \rightarrow T_1)$
Cycle $X(T_1 \rightarrow T_2), YZ\ (T_2 \rightarrow T_3), Y(T_3 \rightarrow T_1)$

Database Recovery

A Transaction may Fail

- Why recovery is needed? (What causes a Transaction to abort)
  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 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.

A Transaction may Fail

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
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

A Transaction may Fail

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, overwriting disks

Transaction State

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

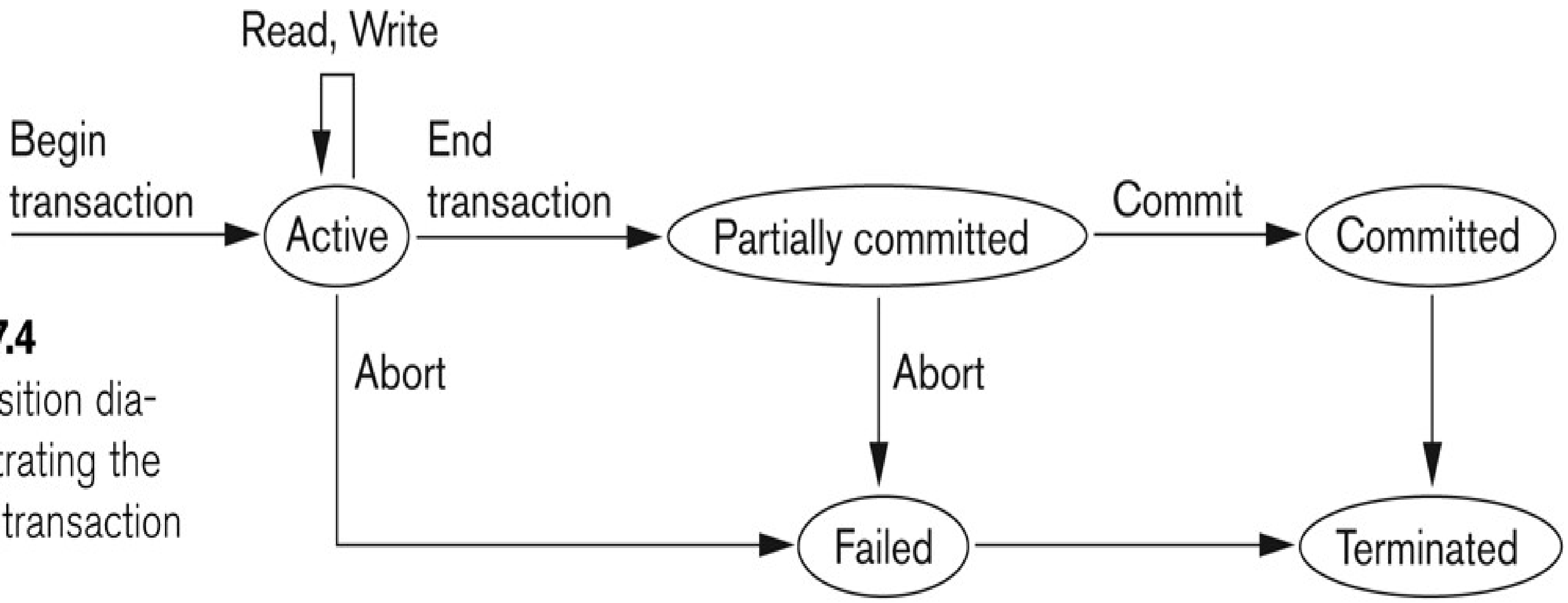


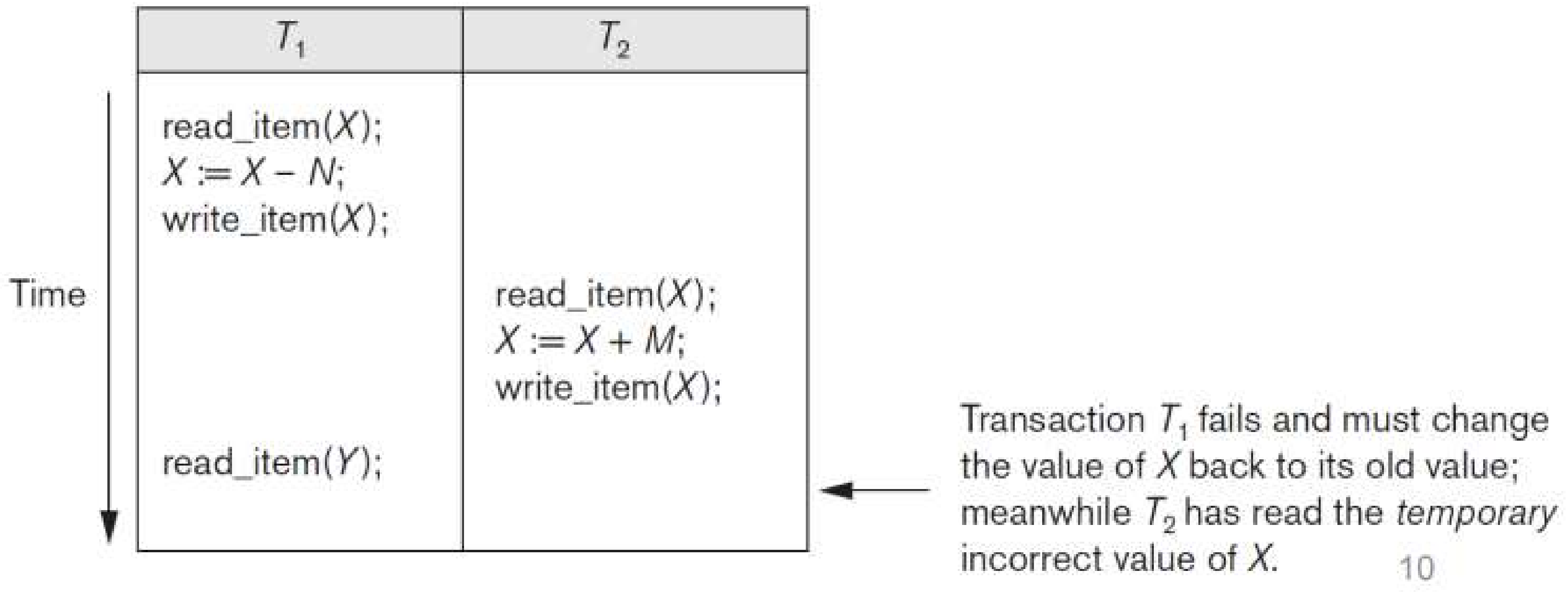
Figure 17.4
State transition diagram illustrating the states for transaction execution.

Undo Logging for Recovery

- A log is a file of log records, each telling something about what some transaction has done
- As transactions execute, the log manager records in the log each important event (e.g., write operations)
- Logs are initially created in main memory and are allocated by the buffer manager
  - Why not on disk? Disk I/O takes a lot of time
- Logs are periodically copied to disk by "flush-log" operation
- If log records appear in nonvolatile storage (disk), we can use them to restore the database to a consistent state after a system crash
- After a system failure, all data in volatile storage (memory) will lose but the data in nonvolatile storage remain

Recoverability Problems in Concurrent Schedule

- Dirty Read Problem (write/read conflicts)
  - ◆ This problem occurs when one transaction updates a database item and then the transaction fails for some reason. Meanwhile, the updated item is accessed (read) by another transaction before it is changed back (or rolled back) to its original value.



T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);	

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

read_item(Y);	

Transaction T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the *temporary* incorrect value of X.

10

Schedules Classified on Recoverability

- Non-Recoverable schedule:
  - The committed transaction with dirty-read problem cannot be rolled back.

- Recoverable schedule:
  - No committed transaction needs to be rolled back.
  - A schedule S is recoverable if

    For all Ti and Tj where Ti read an item written by Tj,

    Tj commits before Ti

Example 1 (Non-Recoverable vs Recoverable)

T1	T2
Read(A)	
A=A+1	
Write(A)	
	Read(A)
	Read(B)
	B=B+A
	…
	Commit/Abort
Commit/Abort	

T1	T2
Read(A)	
A=A+1	
Write(A)	
	Read(A)
	Read(B)
	B=B+A
	…
Commit/Abort	
	Commit/Abort

Non-recoverable

If T2 commits and then

T1 Aborts…

Recoverable

T2 commits after T1

Schedules Classified on Recoverability

- Cascaded rollback:
  - A single rollback leads to a series of rollback
  - All uncommitted transactions that read data items from a failed (aborted) transaction must be rolled back
- Cascadeless schedule:
  - Every transaction reads only the items that are written by committed transactions.
  - In other words

    Before Ti reads an item written by Tj, Tj is already committed
- 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.

Example 2 (Non-Cascadeless vs Cascadeless)

T1	T2
Read(A)	
A=A+1	
Write(A)	
	Read(A)
	Read(B)
	B=B+A
	….
Commit/Abort	
	Commit/Abort

T1	T2
Read(A)	
A=A+1	
Write(A)	
Commit/Abort	
	Read(A)
	Read(B)
	B=B+A
	…
	Commit/Abort

Non-cascadeless

T1 rollbacks, T2 also has to roll back

Cacadeless

T2 read A after T1 commits

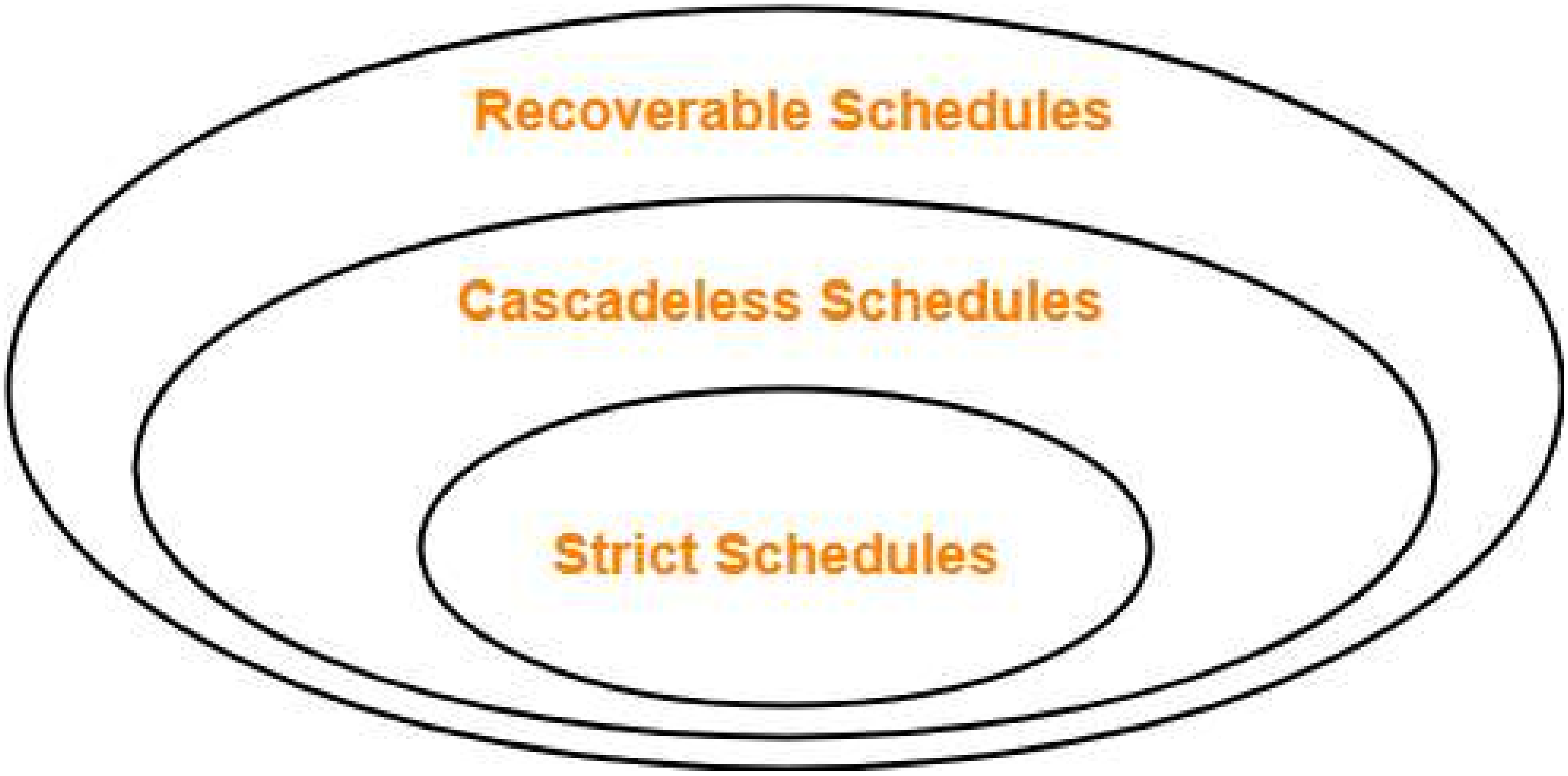
Example 3 (Cascadeless vs Strict)

- **S: w1 (X); w2 (X); a1 (T1 aborts)**
- Initially, X=9
- T1 writes a value 5 for X (keeping 9 as "before image")
- T2 writes a value 8 for X (keeping 5 as "before image")
- Then T1 aborts

- Cascadeless schedule:
  - **If the system restore according to the "before image" kept in T1, then 9 will be the value for X now, which means, the effect of T2 is lost**

- Strict Schedules:
  - **w2 can not happen until T1 commits**

Relationship of recoverable, cascadeless and strict



Relationship between recoverability and serializability

- The concepts of Serializability and Recoverability are ***orthogonal* concepts!**

- A schedule can be:
    1) not serializable and not recoverable
    2) not serializable but recoverable
    3) Serializable but not recoverable
    4) Serializable and recoverable

ACID Properties of Transactions

- Atomicity: A transaction is an atomic unit of processing. It is either performed completely or not performed at all (all or nothing)

- Consistency: A correct execution of a transaction must take the database from one consistent state to another (correctness)

- Isolation: A transaction should not make its updates visible to other transactions until it is committed (no partial results)

- Durability: Once a transaction changes the database state and the changes are committed, these changes must never be lost because of subsequent failure (committed and permanent results)

References

- 6e
  - *Chapter 20, pages 721-750*