Database Systems

CHAPTER 20

Introduction to Transaction Processing Concepts and Theory

Introduction

- Transaction
  - Describes local unit of database processing
- Transaction processing systems
  - Systems with large databases and hundreds of concurrent users
  - Require high availability and fast response time

Introduction to Transaction Processing

- **Single-user DBMS**
  - At most one user at a time can use the system
  - Example: home computer
- **Multiuser DBMS**
  - Many users can access the system (database) concurrently
  - Example: airline reservations system

Introduction to Transaction Processing (cont'd.)

- **Multiprogramming**
  - Allows operating system to execute multiple processes concurrently
  - Executes commands from one process, then suspends that process and executes commands from another process, etc.

Introduction to Transaction Processing (cont'd.)

- Interleaved processing
- Parallel processing
  - Processes C and D in figure below

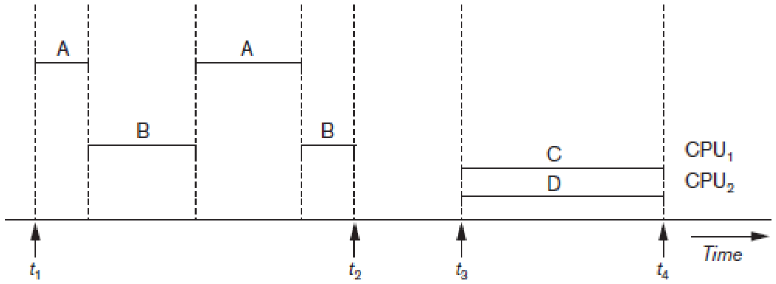


Figure 20.1 Interleaved processing versus parallel processing of concurrent transactions

Transactions

- Transaction: an executing program
  - Forms logical unit of database processing
- Begin and end transaction statements
  - Specify transaction boundaries
- Read-only transaction
- Read-write transaction

Database Items

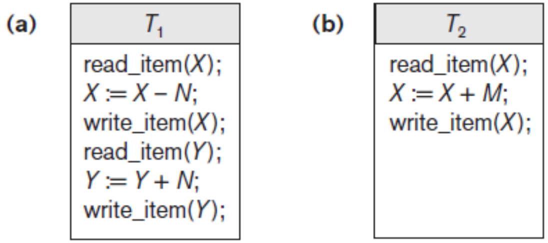
- Database represented as collection of named data items
- Size of a data item called its granularity
- Data item
  - Record
  - Disk block
  - Attribute value of a record
- Transaction processing concepts independent of item granularity

Read and Write Operations

- read_item(X)
  - Reads a database item named X into a program variable named X
  - Process includes finding the address of the disk block, and copying to and from a memory buffer
- write_item(X)
  - Writes the value of program variable X into the database item named X
  - Process includes finding the address of the disk block, copying to and from a memory buffer, and storing the updated disk block back to disk

Read and Write Operations (cont'd.)

- Read set of a transaction
  - Set of all items read
- Write set of a transaction
  - Set of all items written



(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);
```

Figure 20.2 Two sample transactions (a) Transaction *T*1 (b) Transaction *T*2

DBMS Buffers

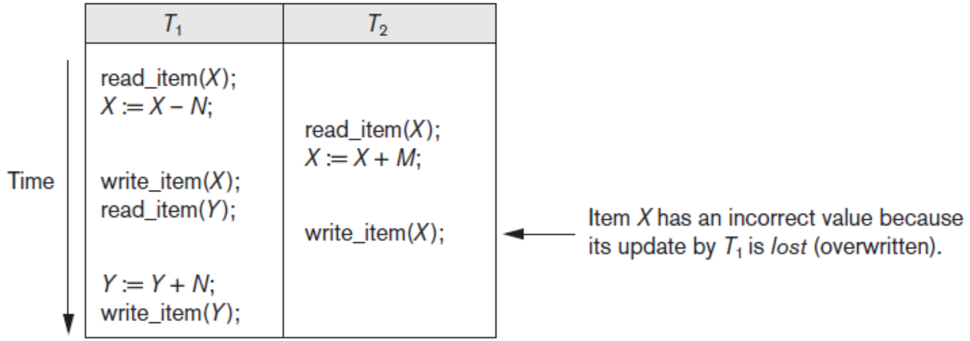
- DBMS will maintain several main memory data buffers in the database cache
- When buffers are occupied, a buffer replacement policy is used to choose which buffer will be replaced
  - Example policy: least recently used

Concurrency Control

- Transactions submitted by various users may execute concurrently
  - Access and update the same database items
  - Some form of concurrency control is needed
- The lost update problem
  - Occurs when two transactions that access the same database items have operations interleaved
  - Results in incorrect value of some database items

The Lost Update Problem

(a)



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

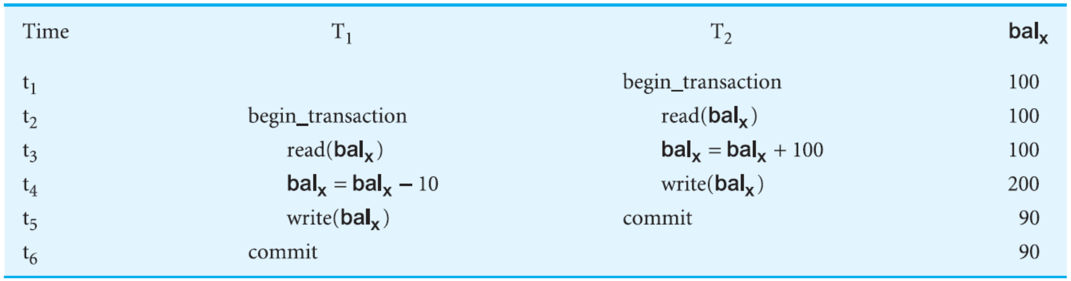
write_item(X);
read_item(Y);	

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

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

Figure 20.3 Some problems that occur when concurrent execution is uncontrolled (a) The lost update problem

The Lost Update Problem

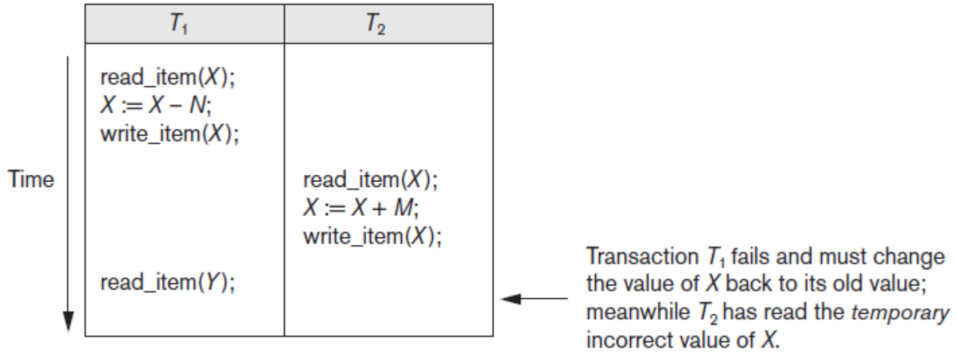


Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	read(bal_x)	100
t_3	read(bal_x)	$bal_x = bal_x + 100$	100
t_4	$bal_x = bal_x - 10$	write(bal_x)	200
t_5	write(bal_x)	commit	90
t_6	commit		90

Figure 20.3 Some problems that occur when concurrent execution is uncontrolled (a) The lost update problem

The Temporary Update Problem OR Dirty Read Problem OR Uncommitted Dependency Problem

(b)



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.

Figure 20.3 (cont'd.) Some problems that occur when concurrent execution is uncontrolled (b) The temporary update problem

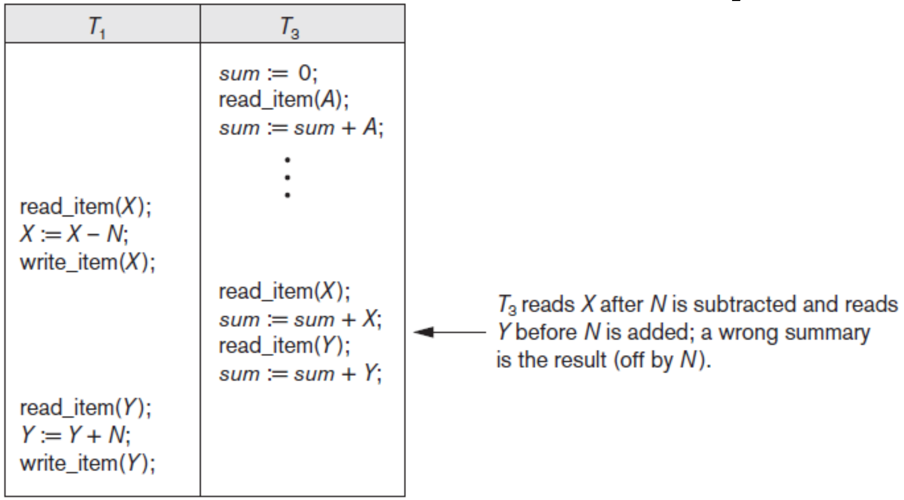
The Temporary Update Problem OR Dirty Read Problem OR Uncommitted Dependency Problem

Time	T_3	T_4	$\mathbf{bal_x}$
t_1		begin_transaction	100
t_2		read($\mathbf{bal_x}$)	100
t_3		$\mathbf{bal_x} = \mathbf{bal_x} + 100$	100
t_4	begin_transaction	write($\mathbf{bal_x}$)	200
t_5	read($\mathbf{bal_x}$)	⋮	200
t_6	$\mathbf{bal_x} = \mathbf{bal_x} - 10$	**rollback**	100
t_7	write($\mathbf{bal_x}$)		190
t_8	commit		190

Figure 20.3 (cont'd.) Some problems that occur when concurrent execution is uncontrolled (b) The temporary update problem

The Incorrect Summary Problem

(c)



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).

Figure 20.3 (cont'd.) Some problems that occur when concurrent execution is uncontrolled (c) The incorrect summary problem

The Incorrect Summary Problem OR Inconsistent Analysis Problem

Time	T_5	T_6	$\mathbf{bal_x}$	$\mathbf{bal_y}$	$\mathbf{bal_z}$	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum = 0	100	50	25	0
t_3	read($\mathbf{bal_x}$)	read($\mathbf{bal_x}$)	100	50	25	0
t_4	$\mathbf{bal_x} = \mathbf{bal_x} - 10$	$\text{sum} = \text{sum} + \mathbf{bal_x}$	100	50	25	100
t_5	write($\mathbf{bal_x}$)	read($\mathbf{bal_y}$)	90	50	25	100
t_6	read($\mathbf{bal_z}$)	$\text{sum} = \text{sum} + \mathbf{bal_y}$	90	50	25	150
t_7	$\mathbf{bal_z} = \mathbf{bal_z} + 10$		90	50	25	150
t_8	write($\mathbf{bal_z}$)		90	50	35	150
t_9	commit	read($\mathbf{bal_z}$)	90	50	35	150
t_{10}		$\text{sum} = \text{sum} + \mathbf{bal_z}$	90	50	35	185
t_{11}		commit	90	50	35	185

Figure 20.3 (cont'd.) Some problems that occur when concurrent execution is uncontrolled (c) The incorrect summary problem

The Unrepeatable Read Problem (Fuzzy Read)

- Transaction T reads the same item twice
- Value is changed by another transaction T′ between the two reads
- T receives different values for the two reads of the same item

The Phantom Read Problem

- Transaction T executes a query that retrieves a set of tuples from a relation satisfying a certain predicate
- T re-executes the query at a later time, but finds that the retrieved set contains an additional (phantom) that has been inserted by another transaction in the meantime.
- T receives different tuples for the same query

Why Recovery is Needed

- Committed transaction
  - Effect recorded permanently in the database
- Aborted transaction
  - Does not affect the database
- Types of transaction failures
  - Computer failure (system crash)
  - Transaction or system error
  - Local errors or exception conditions detected by the transaction

Why Recovery is Needed (cont’d.)

- Types of transaction failures (cont’d.)
  - Concurrency control enforcement
  - Disk failure
  - Physical problems or catastrophes
- System must keep sufficient information to recover quickly from the failure
  - Disk failure or other catastrophes have long recovery times

Transaction and System Concepts

- System must keep track of when each transaction starts, terminates, commits, and/or aborts
  - BEGIN_TRANSACTION
  - READ or WRITE
  - END_TRANSACTION
  - COMMIT_TRANSACTION
  - ROLLBACK (or ABORT)

Transaction and System Concepts (cont'd.)

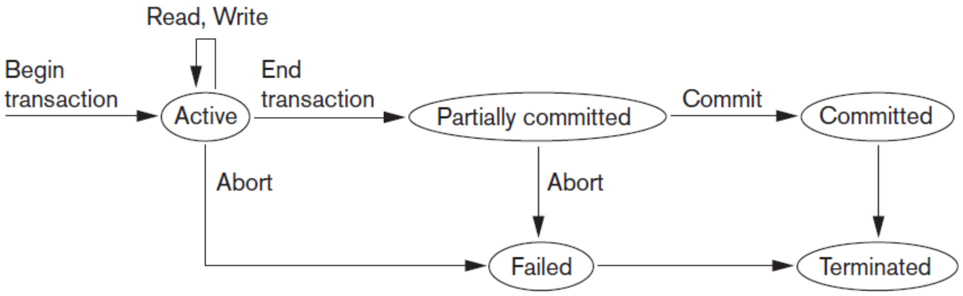


Figure 20.4 State transition diagram illustrating the states for transaction execution

The System Log

- System log keeps track of transaction operations
- Sequential, append-only file
- Not affected by failure (except disk or catastrophic failure)
- Log buffer
  - Main memory buffer
  - When full, appended to end of log file on disk
- Log file is backed up periodically
- Undo and redo operations based on log possible

Commit Point of a Transaction

- Occurs when all operations that access the database have completed successfully
  - And effect of operations recorded in the log
- Transaction writes a commit record into the log
  - If system failure occurs, can search for transactions with recorded start_transaction but no commit record
- Force-writing the log buffer to disk
  - Writing log buffer to disk before transaction reaches commit point

DBMS-Specific Buffer Replacement Policies

- Page replacement policy
  - Selects particular buffers to be replaced when all are full
- Domain separation (DS) method
  - Each domain handles one type of disk pages
    - Index pages
    - Data file pages
    - Log file pages
  - Number of available buffers for each domain is predetermined

DBMS-Specific Buffer Replacement Policies (cont'd.)

- Hot set method
  - Useful in queries that scan a set of pages repeatedly
  - Does not replace the set in the buffers until processing is completed
- The DBMIN method
  - Predetermines the pattern of page references for each algorithm for a particular type of database operation
  - some queries may reference the same file twice, so there would be a locality set for each file instance needed in the query
    - Calculates locality set using Query Locality Set Model (QLSM)

DBMS-Specific Buffer Replacement Policies (cont'd.)

DBMin

- Buffers are managed on a **per file instance**
  - Active instances of the same file are given different buffer pools – and *may use different replacement policies!*
  - Files **share pages** via global table
- Set of buffered pages associated with a file instance is called its **locality set**

Desirable Properties of Transactions

- ACID properties
  - Atomicity
    - Transaction performed in its entirety or not at all
  - Consistency preservation
    - Takes database from one consistent state to another
  - Isolation
    - Not interfered with by other transactions
  - Durability or permanency
    - Changes must persist in the database

Desirable Properties of Transactions (cont'd.)

- Levels of isolation
  - Level 0 isolation does not overwrite the dirty reads of higher-level transactions
  - Level 1 isolation has no lost updates
  - Level 2 isolation has no lost updates and no dirty reads
  - Level 3 (true) isolation has repeatable reads
    - In addition to level 2 properties
  - Snapshot isolation
    - transaction sees the data items that it reads based on the committed values of the items in the database snapshot (or database state) when the transaction starts. Snapshot isolation will ensure that the phantom record problem does not occur

Characterizing Schedules Based on Recoverability

- Schedule or history
  - Order of execution of operations from all transactions
  - Operations from different transactions can be interleaved in the schedule
- Total ordering of operations in a schedule
  - For any two operations in the schedule, one must occur before the other

Some Definitions

- Schedule
    - A sequence of the operations by a set of concurrent transactions that preserves the order of the operations in each of the individual transactions.
- Serial Schedule
    - A schedule where the operations of each transaction are executed consecutively without any interleaved operations from other transactions.
- Non-serial Schedule
    - A schedule where the operations from a set of concurrent transactions are interleaved.

Serializablity

- **Serializability**
  - Find non-serial schedules that allow transactions to execute concurrently without interfering with one another, and thereby produce a database state that could be produced by a serial execution.
- If a set of transactions executes concurrently,
- We say that the (non-serial) schedule is correct
  - If it produces the same result as some serial execution.
- Such a schedule is called ***serializable***.
- To prevent inconsistency from transactions interfering with one another… it is essential to guarantee serializability of concurrent transactions.

Characterizing Schedules Based on Recoverability (cont'd.)

- Two conflicting operations in a schedule
  - Operations belong to different transactions
  - Operations access the same item X
  - At least one of the operations is a write_item(X)
- Two operations conflict if changing their order results in a different outcome
- Read-write conflict
- Write-write conflict

Characterizing Schedules Based on Recoverability (cont'd.)

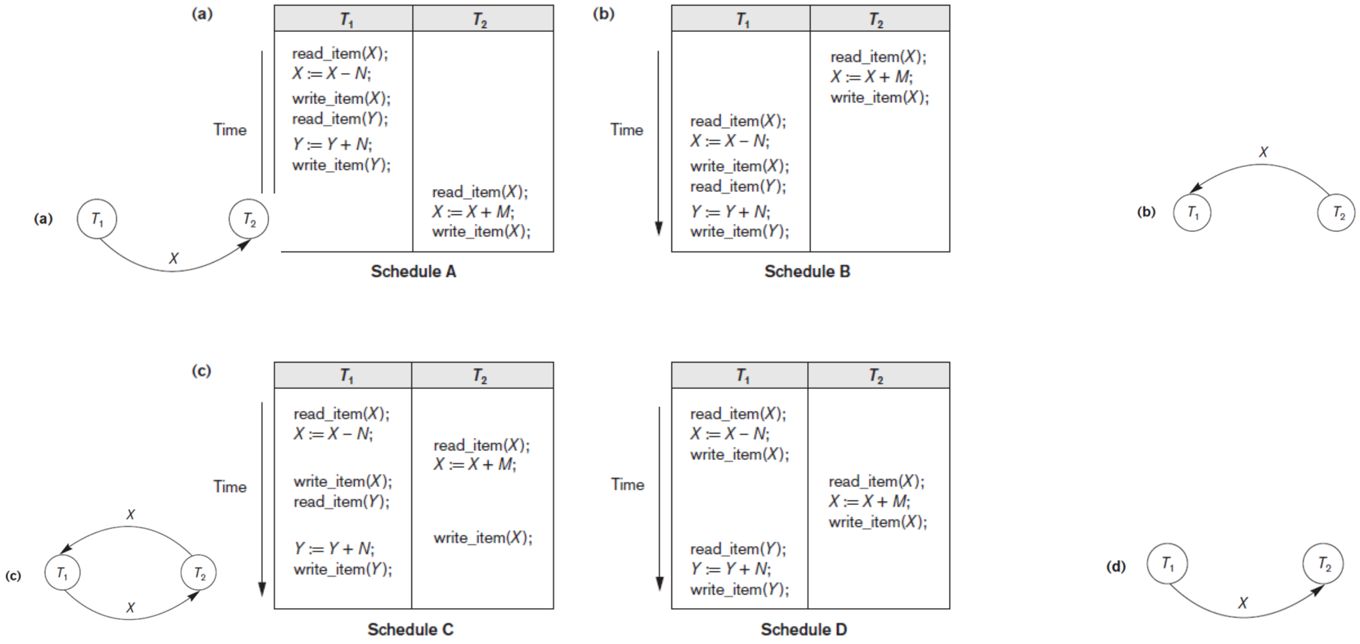
- **Recoverable schedules**
  - Recovery is possible
- Nonrecoverable schedules should not be permitted by the DBMS
- No committed transaction ever needs to be rolled back
- ***Cascading rollback*** is a situation: where an uncommitted transaction has to be rolled back because it read an item from a transaction that failed.
  - Cascading rollback may occur in some recoverable schedules

Characterizing Schedules Based on Recoverability (cont’d.)

- **Cascadeless schedule**
  - Avoids cascading rollback
- **Strict schedule**
  - Transactions can neither read nor write an item X until the last transaction that wrote X has committed or aborted
  - Simpler recovery process
    - Restore the before image

Characterizing Schedules Based on Serializability

- Serializable schedules
  - Always considered to be correct when concurrent transactions are executing
  - Places simultaneous transactions in series
    - Transaction T_1 before T_2, or vice versa



Schedule 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);

$$\begin{tikzcd} T_1 \arrow[r, bend right, "X"'] & T_2 \end{tikzcd}$$

Schedule 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);	

$$\begin{tikzcd} T_1 & T_2 \arrow[l, bend right, "X"'] \end{tikzcd}$$

Schedule 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);	

$$\begin{tikzcd} T_1 \arrow[r, bend right, "X"'] & T_2 \arrow[l, bend right, "X"'] \end{tikzcd}$$

Schedule D

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);	

$$\begin{tikzcd} T_1 \arrow[r, bend right, "X"'] & T_2 \end{tikzcd}$$

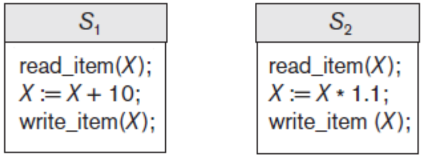
Figure 20.5 Examples of serial and nonserial schedules involving transactions *T*1 and *T*2 (a) Serial schedule A: *T*1 followed by *T*2 (b) Serial schedule B: *T*2 followed by *T*1 (c) Two nonserial schedules C and D with interleaving of operations

Characterizing Schedules Based on Serializability (cont’d.)

- Problem with serial schedules
  - Limit concurrency by prohibiting interleaving of operations
  - Unacceptable in practice
  - Solution: determine which schedules are equivalent to a serial schedule and allow those to occur
- Serializable schedule of *n* transactions
  - Equivalent to some serial schedule of same *n* transactions

Characterizing Schedules Based on Serializability (cont'd.)

- Result equivalent schedules
  - Produce the same final state of the database
    - May be accidental
  - Cannot be used alone to define equivalence of schedules
    - They execute different transactions



S_1	S_2

read_item(X);
$X := X + 10$;
write_item(X);
	read_item(X);
	$X := X * 1.1$;
	write_item (X);

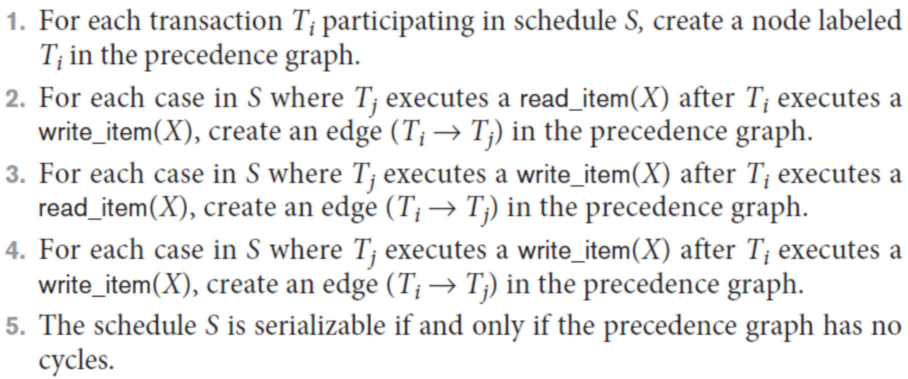
Figure 20.6 Two schedules that are result equivalent for the initial value of *X* = 100 but are not result equivalent in general

Characterizing Schedules Based on Serializability (cont’d.)

- Conflict equivalence
  - Relative order of any two conflicting operations is the same in both schedules
- Serializable schedules (Called conflict serializable)
  - Schedule S is serializable if it is conflict equivalent to some serial schedule S’.

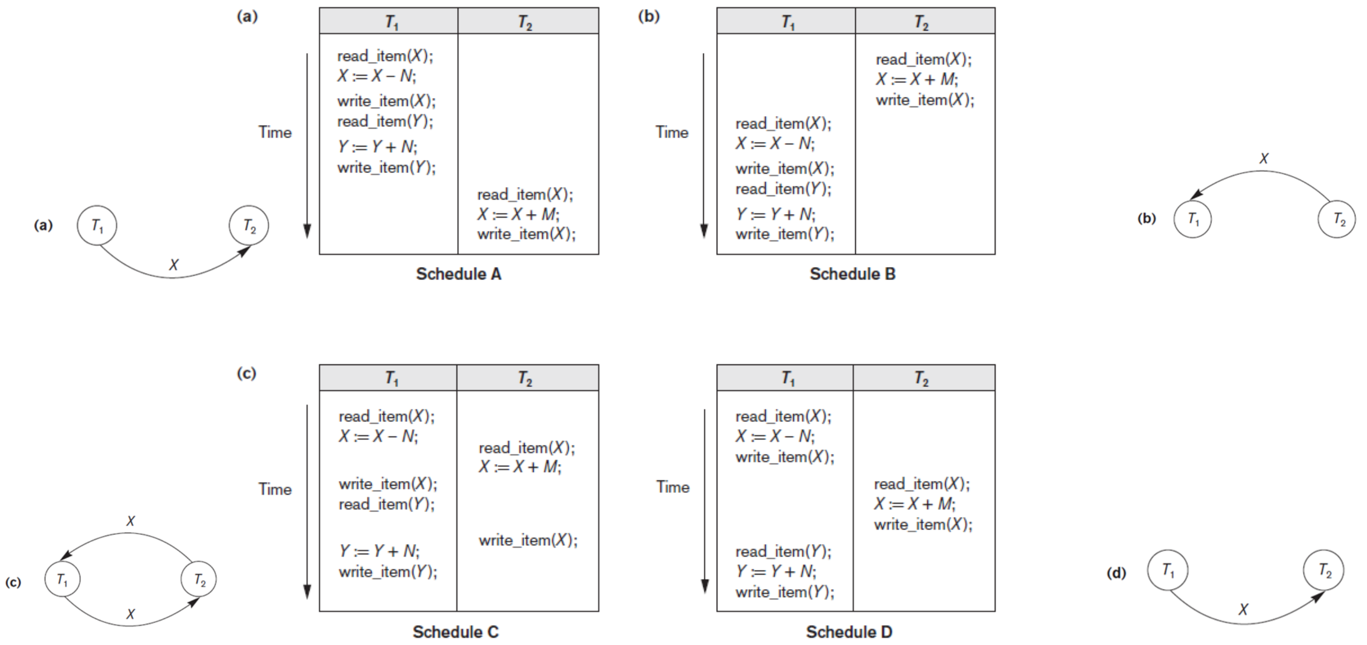
Characterizing Schedules Based on Serializability (cont'd.)

- Testing for serializability of a schedule



1. For each transaction T_i participating in schedule S, create a node labeled T_i in the precedence graph.
2. For each case in S where T_j executes a read_item(X) after T_i executes a write_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
3. For each case in S where T_j executes a write_item(X) after T_i executes a read_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
4. For each case in S where T_j executes a write_item(X) after T_i executes a write_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
5. The schedule S is serializable if and only if the precedence graph has no cycles.

Algorithm 20.1 Testing conflict serializability of a schedule S



(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 20.7 Constructing the precedence graphs for schedules A to D from Figure 20.5 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 serializable) (d) Precedence graph for schedule D (serializable, equivalent to schedule A)

How Serializability is Used for Concurrency Control

- Being serializable is different from being serial
- Serializable schedule gives benefit of concurrent execution
  - Without giving up any correctness
- Difficult to test for serializability in practice
  - Factors such as system load, time of transaction submission, and process priority affect ordering of operations
- DBMS enforces protocols
  - Set of rules to ensure serializability

View Equivalence and View Serializability

- View equivalence of two schedules
  - 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 must produce the same results
  - Read operations said to see the same view in both schedules
- View serializable schedule
  - View equivalent to a serial schedule

(a)

Transaction T_1
read_item(X);
write_item(X);
read_item(Y);
write_item(Y);

Transaction T_2
read_item(Z);
read_item(Y);
write_item(Y);
read_item(X);
write_item(X);

Transaction T_3
read_item(Y);
read_item(Z);
write_item(Y);
write_item(Z);

(b)

Time ↓

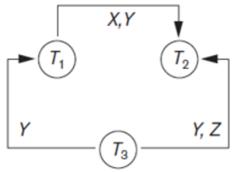
Transaction T_1	Transaction T_2	Transaction T_3
	read_item(Z); read_item(Y); write_item(Y);	
		read_item(Y); read_item(Z);
read_item(X); write_item(X);		
		write_item(Y); write_item(Z);
	read_item(X);	
read_item(Y); write_item(Y);		
	write_item(X);	

Schedule E

(d)

$$\begin{tikzcd}
T_1 \arrow[rr, bend right, "X"'] & & T_2 \arrow[ll, bend right, "Y"'] \arrow[dl, "{Y, Z}"] \\
& T_3 \arrow[ul, "Y"] &
\end{tikzcd}$$

(e)



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

(c)

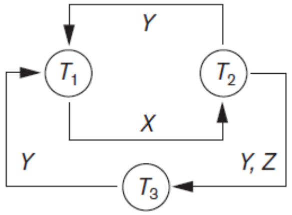
Time

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

Schedule F

Figure 20.8 (continued)
Another example of serializability testing. (d) Precedence graph for schedule E. (e) Precedence graph for schedule F. (f) Precedence graph with two equivalent serial schedules.

(d)



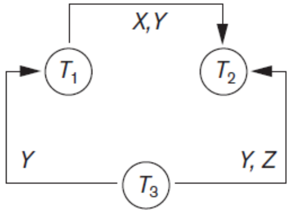
Equivalent serial schedules

None

Reason

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

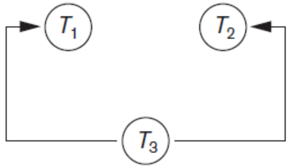
(e)



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

(f)



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

$T_3 \rightarrow T_2 \rightarrow T_1$

View Equivalent

Two schedules T1 and T2 are said to be view equivalent, if they satisfy all the following conditions:

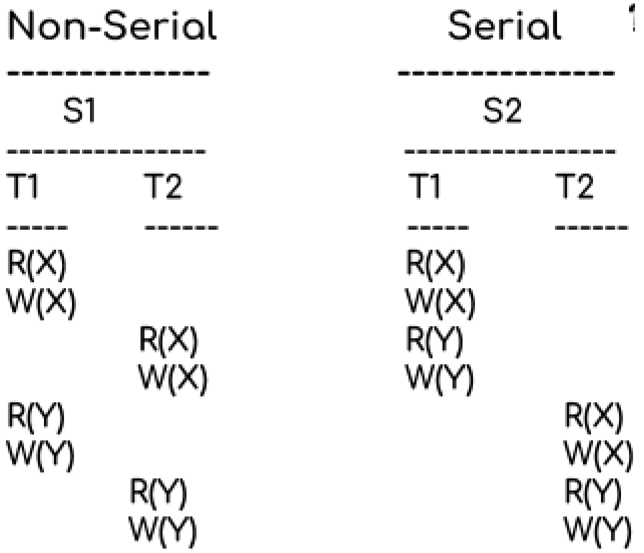
1. **Initial Read:** Initial read of each data item in transactions must match in both schedules. For example, if transaction T1 reads a data item X before transaction T2 in schedule S1 then in schedule S2, T1 should read X before T2.

- Read vs Initial Read: You may be confused by the term initial read. Here initial read means the first read operation on a data item, for example, a data item X can be read multiple times in a schedule but the first read operation on X is called the initial read.

2**. Final Write:** Final write operations on each data item must match in both the schedules. For example, a data item X is last written by Transaction T1 in schedule S1 then in S2, the last write operation on X should be performed by the transaction T1.

Update Read: If in schedule S1, the transaction T1 is reading a data item updated by T2 then in schedule S2, T1 should read the value after the write operation of T2 on same data item. For example, In schedule S1, T1 performs a read operation on X after the write operation on X by T2 then in S2, T1 should read the X after T2 performs write on X.

View Serializable schedule



Non-Serial		Serial	
S1		S2	
T1	T2	T1	T2
R(X)		R(X)	
W(X)		W(X)	
	R(X)	R(Y)	
	W(X)	W(Y)	
R(Y)			R(X)
W(Y)			W(X)
	R(Y)		R(Y)
	W(Y)		W(Y)