Introduction to Transaction Processing Concepts and Theory

406.426 Design & Analysis of Database Systems

Jonghun Park

jonghun@snu.ac.kr

Dept. of Industrial Engineering

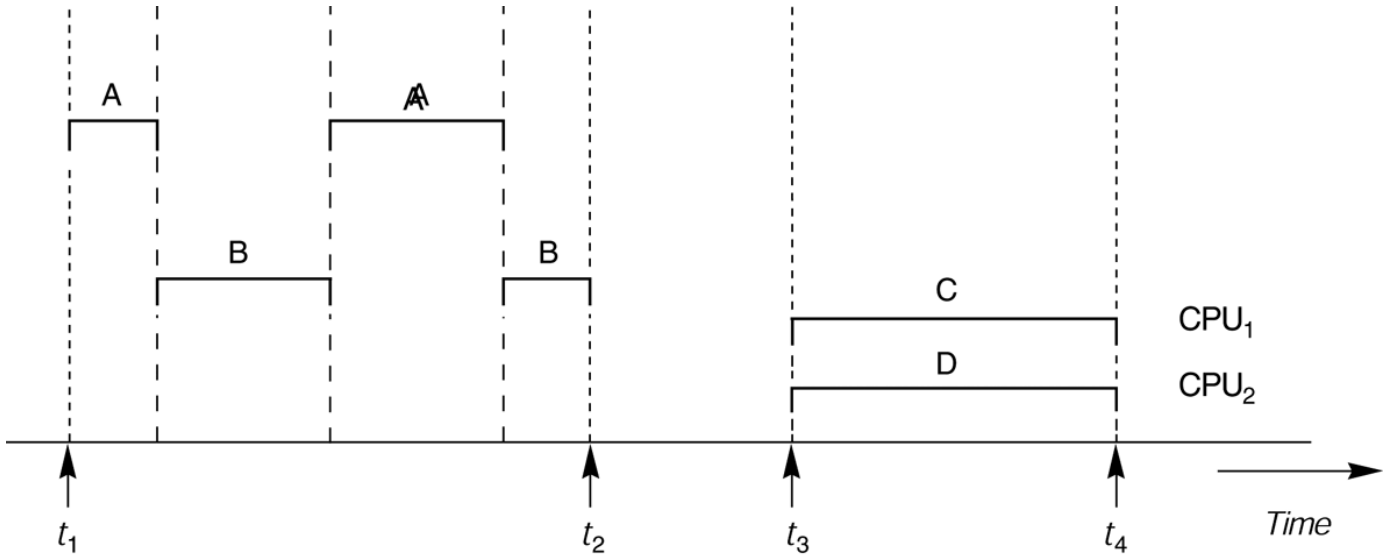
Seoul National University

chapter outline

- Introduction to Transaction Processing
- Transaction and System Concepts
- Desirable Properties of Transactions
- Characterizing Schedules based on Recoverability
- Characterizing Schedules based on Serializability
- Transaction Support in SQL

single-user vs. multi-user systems

- concurrency
- single-user, multi-user
- multi-programming
- interleaving vs. parallel processing
- resources that are concurrently accessed: the stored data items



Transactions, and Read / Write Operations

- transaction
  - an executing program that forms a **logical unit** of DB processing
  - includes **one or more DB access operations**
- transaction boundaries
  - “begin transaction” and “end transaction”
- in TP, a DB is basically represented as a collection of **named data items**
- a transaction includes **read_item** and **write_item** operations to access and update the DB
  - read_item(X): reads a DB item named X into a program variable
  - write_item(X): writes the value of program variable X into the DB item named X

2 sample transactions

- named data item: # of reserved seats
- (a) T_1 transfers N reservations from one flight whose # of reserved seats is stored in the database item named X to another flight whose # of reserved seats is stored in the database item named Y
- (b) T_2 reserves M seats on the first flight (X) referenced in transaction T_1

(a)

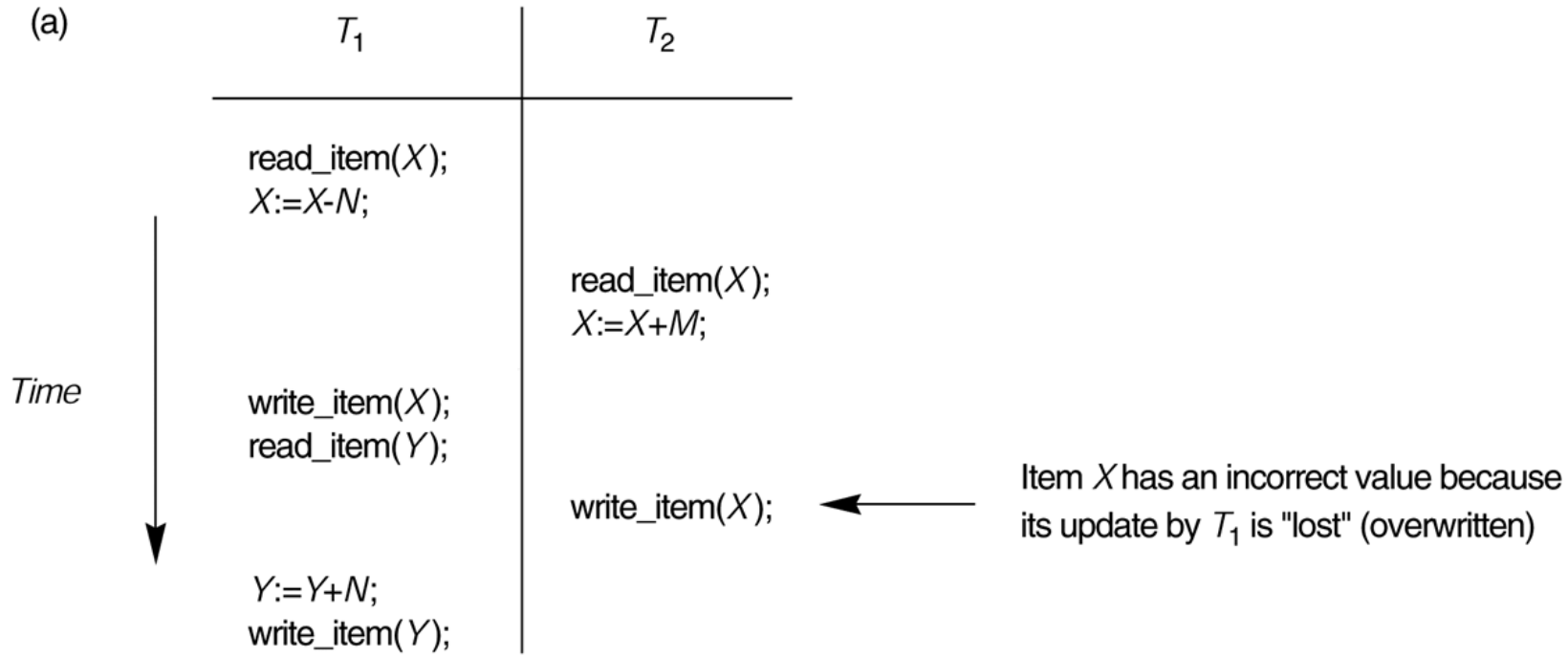
T_1
read_item (X);
X:=X-N;
write_item (X);
read_item (Y);
Y:=Y+N;
write_item (Y);

(b)

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

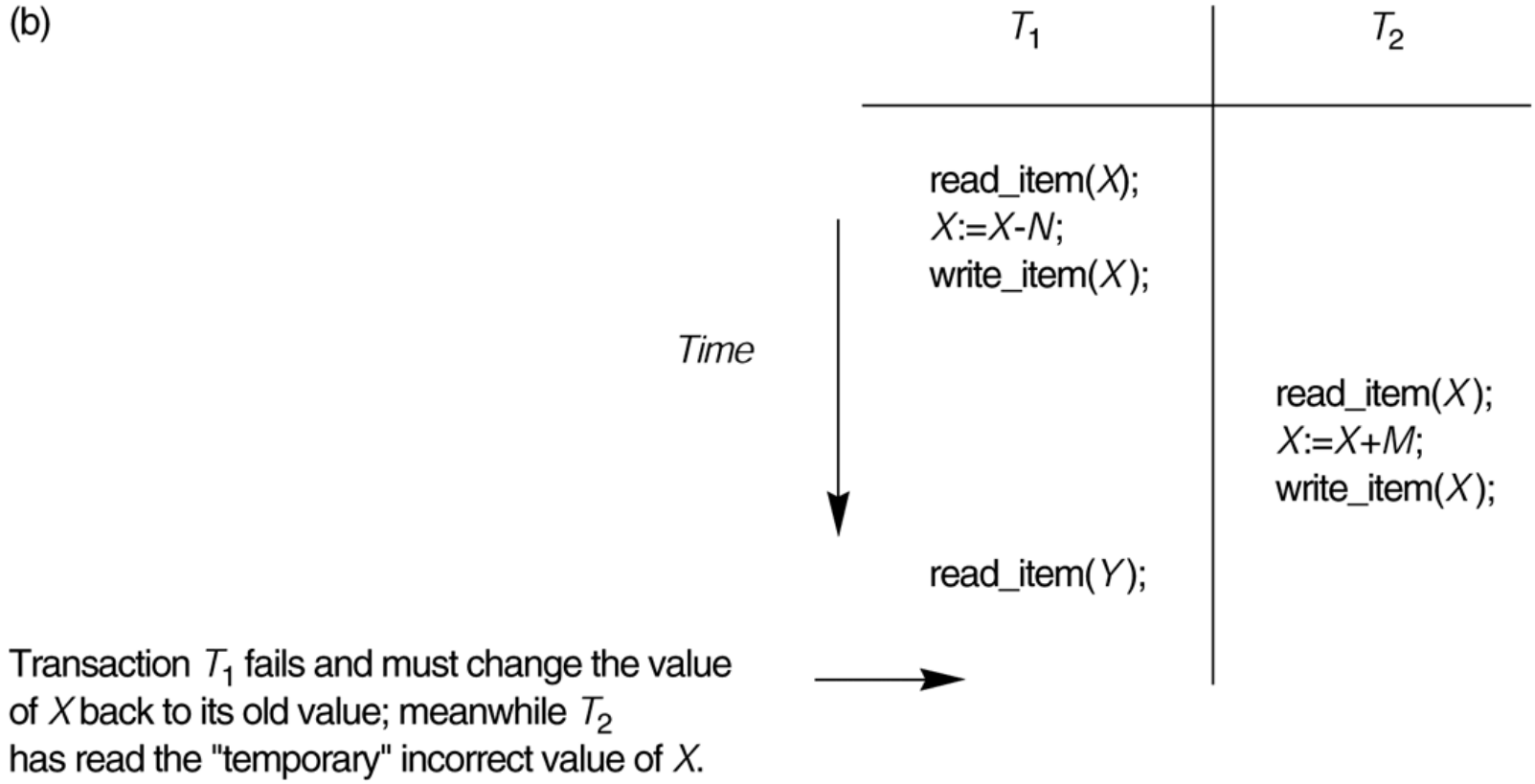
motivation for concurrency control

- **lost update problem**
  - when two transactions that access the same database items have their operations interleaved in a way that **makes the value of some database items incorrect**



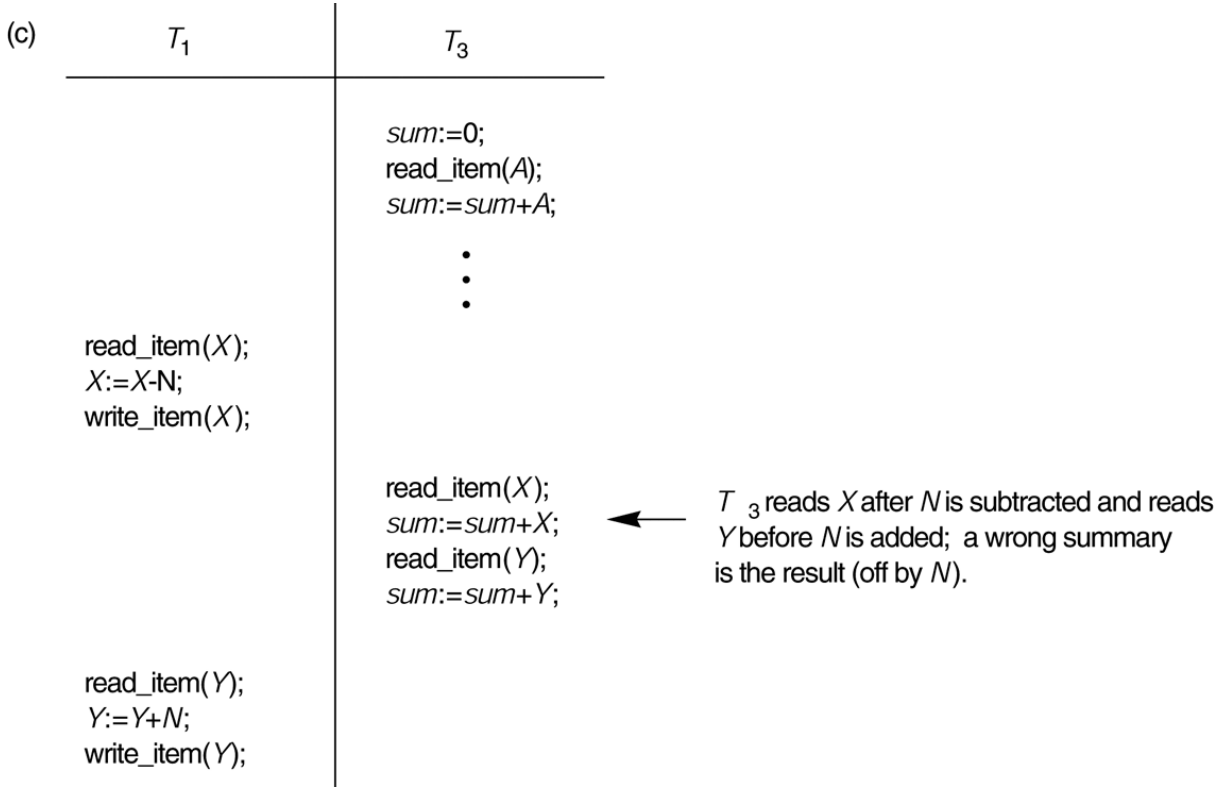
motivation for concurrency control

- temporary update (**dirty read**) problem
  - when one transaction updates a database item and then the transaction fails for some reason



motivation for concurrency control

- **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



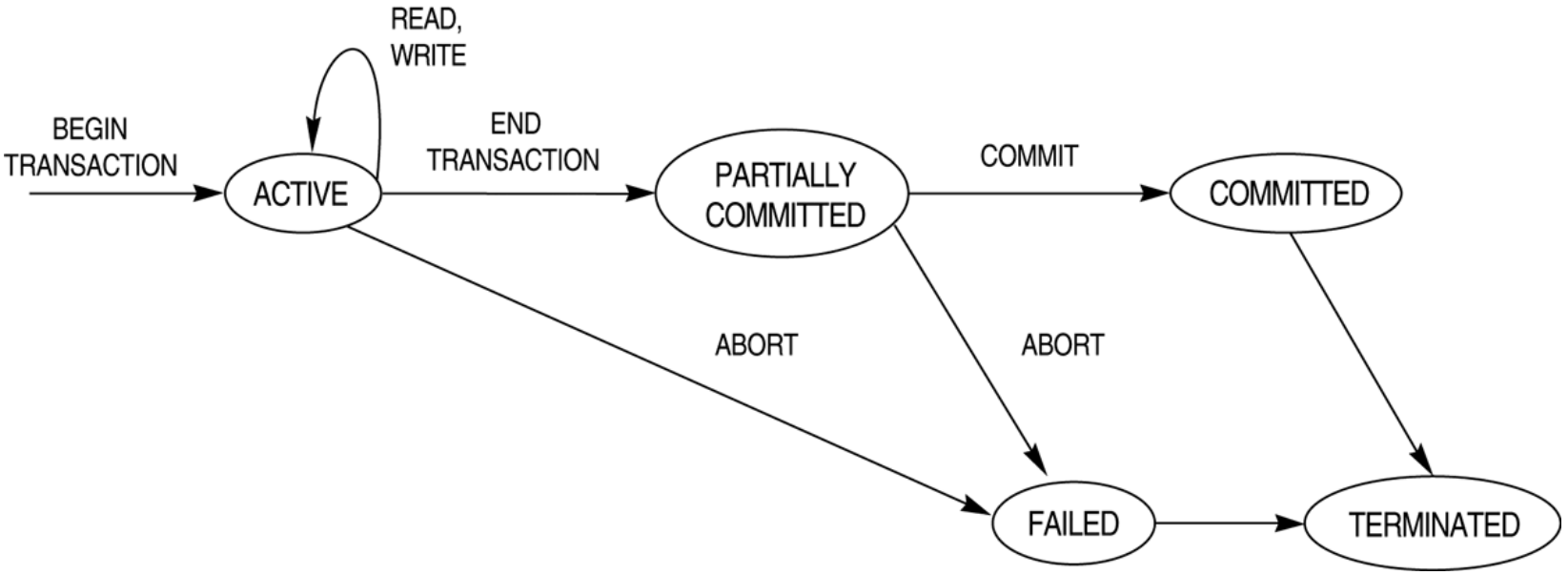
why recovery is needed

- whenever a transaction is submitted to a DBMS for execution, the system is responsible for making sure that
  - either (1) all the operations in the transaction are completed successfully and their effect is **recorded permanently** in the DB,
  - or (2) the transaction has **no effect** whatsoever on the DB or on any other transactions
- types of failures
  - computer failure (system crash)
  - transaction or system error: e.g., division by zero
  - local errors or exception conditions detected by the transaction: e.g., data not found
  - concurrency control enforcement: e.g., abort enforcement
  - disk failure
  - physical problems and catastrophes: e.g., theft

transaction states and additional operations

- transaction is an **atomic unit of work** that is either completed in its entirety or not done at all
- recovery manager keeps track of the following operations:
  - BEGIN_TRANSACTION, READ, WRITE, END_TRANSACTION, COMMIT_TRANSACTION, ROLLBACK (or ABORT)

transaction states and additional operations



- at the “Partially Committed” state, some recovery protocols need to ensure that a system failure will not result in an inability to record the changes of the transaction permanently
- this is usually done by recording changes in the system log

system log

- to be able to recover from failures that affect transactions, the system maintains a log to **keep track of all transaction operations** that affect the values of database items
- if the system crashes, we can recover to a consistent DB state by examining the log and using one of the recovery methods
- types of log entries
  - [start_transaction, T]
  - [write_item, T, X, old_value, new_value]
  - [read_item, T, X]
  - [commit, T]
  - [abort, T]

commit point of a transaction

- 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 have 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
- transaction then writes a **[commit, *T*]** into the log
- if a system failure occurs, we search back in the log for all transactions *T* that have written a [start_transaction, T] into the log but have not written their [commit, *T*] record yet
- these transactions may have to be **rolled back** to undo their effect on the DB during the recovery process

desirable properties of transactions

- ACID properties
  - **atomicity**: a transaction is either performed in its entirety or not performed at all -> responsibility of the recovery system
  - **consistency**: complete execution of a transaction takes the DB from one consistent state to another -> responsibility of the programmer
  - **isolation**: execution of a transaction should not be interfered with by any other transactions executing concurrently -> responsibility of the concurrency control system
  - **durability**: changes applied to the DB by a committed transaction must **persist** in the DB (i.e., should not be lost) -> responsibility of the recovery system

schedules of transactions

- a schedule S of n transactions T_1, T_2, …, T_n is an **ordering** of the operations of the transactions subject to the **constraint** that
  - for each transaction T_i that participates in S, the operations of T_i in S must appear in the **same order** in which they occur in T_i
  - note: operations from other transactions T_j can be **interleaved** with the operations of T_i in S
- notation
  - r: read_item, w: write_item, c: commit, a: abort

example

- S_a: $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$
- S_b: $r_1(X)$; $w_1(X)$; $r_2(X)$; $w_2(X)$; $r_1(Y)$; 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);	

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);	

more on schedules

- **2 operations** in a schedule are said to **conflict** if they satisfy the following conditions
  - they belong to **different transactions**
  - they access the **same item** X
  - **at least one** of the operations is a **write_item(X)**
- examples
  - $r_1(X)$ and $w_2(X)$ conflict
  - $r_1(X)$ and $r_2(X)$ do not conflict
- a schedule S of n transactions T_1, T_2, …, T_n, is said to be a **complete** schedule if the following conditions hold
  - operations in S are exactly those operations in T_1, T_2, …, T_n, including a **commit** or **abort** operation as the last operation for each transaction in S
  - for any pair of operations from the **same** transaction T_i, their **order** of appearance in S is the **same** as their order of appearance in T_i
  - for **any two conflicting operations**, one of the two must occur before the other in S consistently (i.e., w.r.t. T_i) -> allows for a partial order among the nonconflicting operations

recoverability of schedules

- transaction *T* **reads** from transaction *T'* in a schedule *S* if **some** item *X* is first written by *T'* and later read by *T*
- 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**
  - *T'* must not have been aborted before *T* reads item *X*
  - there should be no transactions that write *X* after *T'* writes it and before *T* reads it
- examples
  - S'_a: $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; c_2; $w_1(Y)$; c_1; => recoverable
  - S_c: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; c_2; a_1;
    - T_2 reads item *X* from T_1, and then T_2 commits before T_1 commits => unrecoverable
    - c_2 must be postponed until after T_1 commits => S_d: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; c_1; c_2; is recoverable
    - if T_1 aborts instead of committing, then T_2 should also abort => e.g., S_e: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; a_1; a_2;

cascadeless schedule

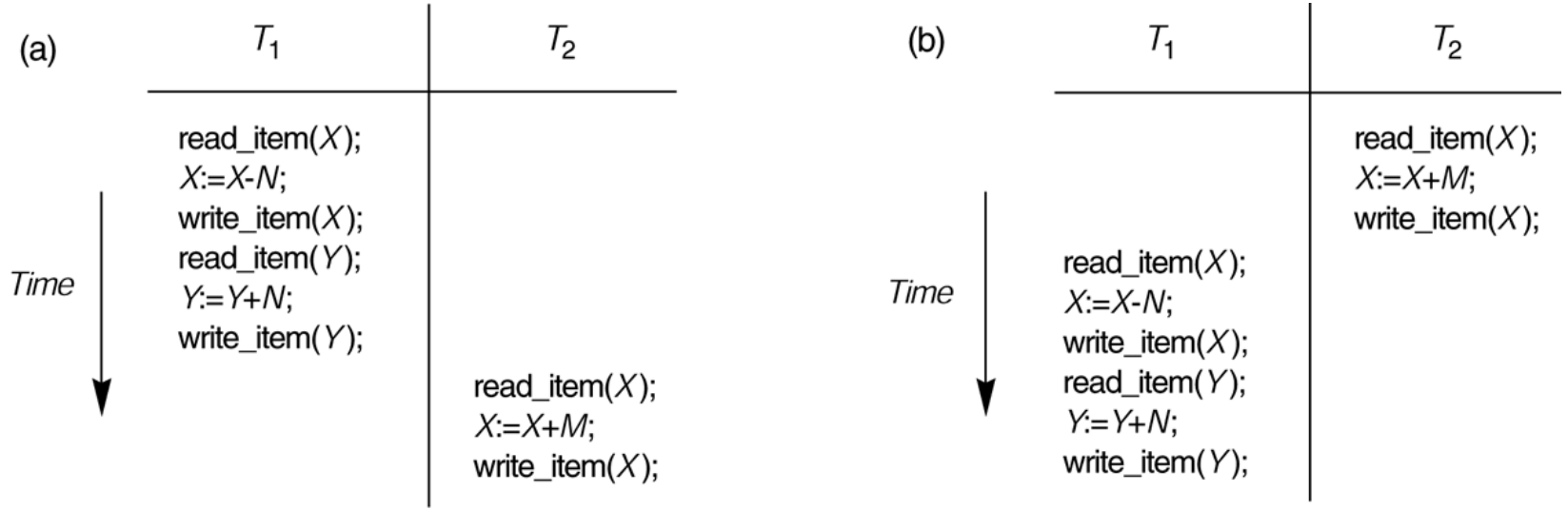
- cascading rollback: an uncommitted transaction has to be rolled back because it read an item from a transaction that is aborted
- cascadeless schedule: every transaction in a schedule **reads** only items that were **written by committed transactions**
- example
  - S_d: $r_1(X)$; $w_1(X)$; $\mathbf{r_2(X)}$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; $\mathbf{c_1}$; c_2; is not cascadeless
    => $r_1(X)$; $w_1(X)$; $r_1(Y)$; $w_1(Y)$; c_1; $\mathbf{r_2(X)}$; $\mathbf{w_2(X)}$; c_2; is cascadeless

strict schedule

- strict schedule
  - transactions can **neither read nor write** an item *X* until the last transaction that **wrote *X*** has **committed** (or **aborted**)
- process of undoing a write_item(*X*) operation of an aborted transaction is simply to restore the **before-image of** data item *X*
- example
  - S_f: $w_1(X,5)$; $w_2(X,8)$; a_1; => cascadeless, but not strict
- implications
  - strict schedule => cascadeless schedule
  - cascadeless schedule => recoverable schedule

serial schedules

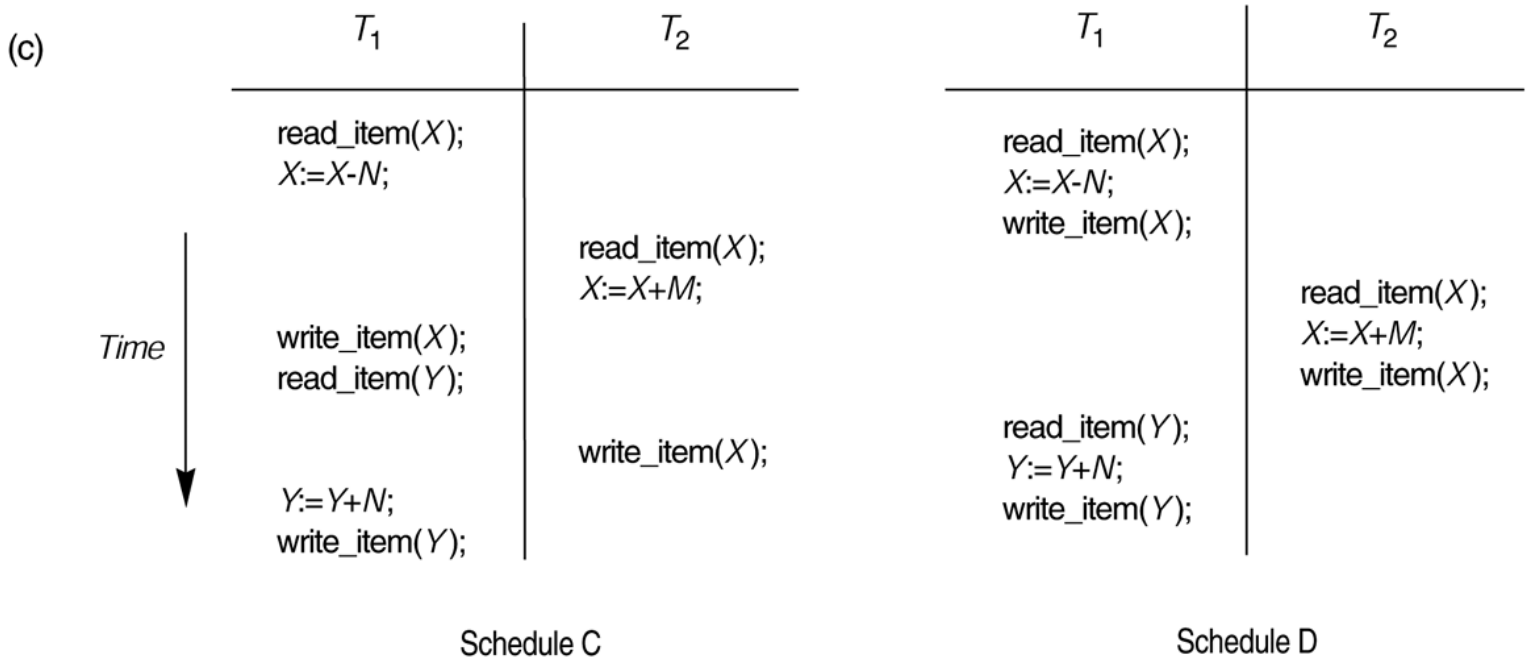
- schedule *S* is **serial** if, for every transaction *T* participating in the schedule, all the operations of *T* are executed consecutively in the schedule **without interleaving**; o.w. nonserial
- if we consider the transactions to be independent, every serial schedule is correct
- example: two airline reservation clerks submit to the DBMS transactions T_1 and T_2



problems with the serial schedules

- limit concurrency or interleaving of operations
  - if a transaction waits for an I/O operation to complete, we cannot switch the CPU to another transaction, thus wasting the CPU time
  - if some transaction *T* is quite long, the other transaction must wait for *T* to complete all its operations before commencing
    - e.g., what if a transaction involves the credit card charge operation which happens to fail?
- unacceptable in practice

nonserial schedules



- example
  - $X = 90$, $Y = 90$, $N = 3$, $M = 2$
  - serial schedule: $X = 89$, $Y = 93$
  - schedule C: $X = 92$, $Y = 93$
  - schedule D: $X = 89$, $Y = 93$
- some nonserial schedules give the correct result => which of the nonserial schedules always give a correct result and which may give erroneous results?

serializability

- a schedule S of n transactions is **serializable** if it is **equivalent** to **some** serial schedule of the same n transactions
  - trivially, serial schedule is serializable
- 2 schedules are called **result equivalent** if they produce the same final state of the database
- in the example below, S_1 and S_2 are result equivalent if $X = 100$

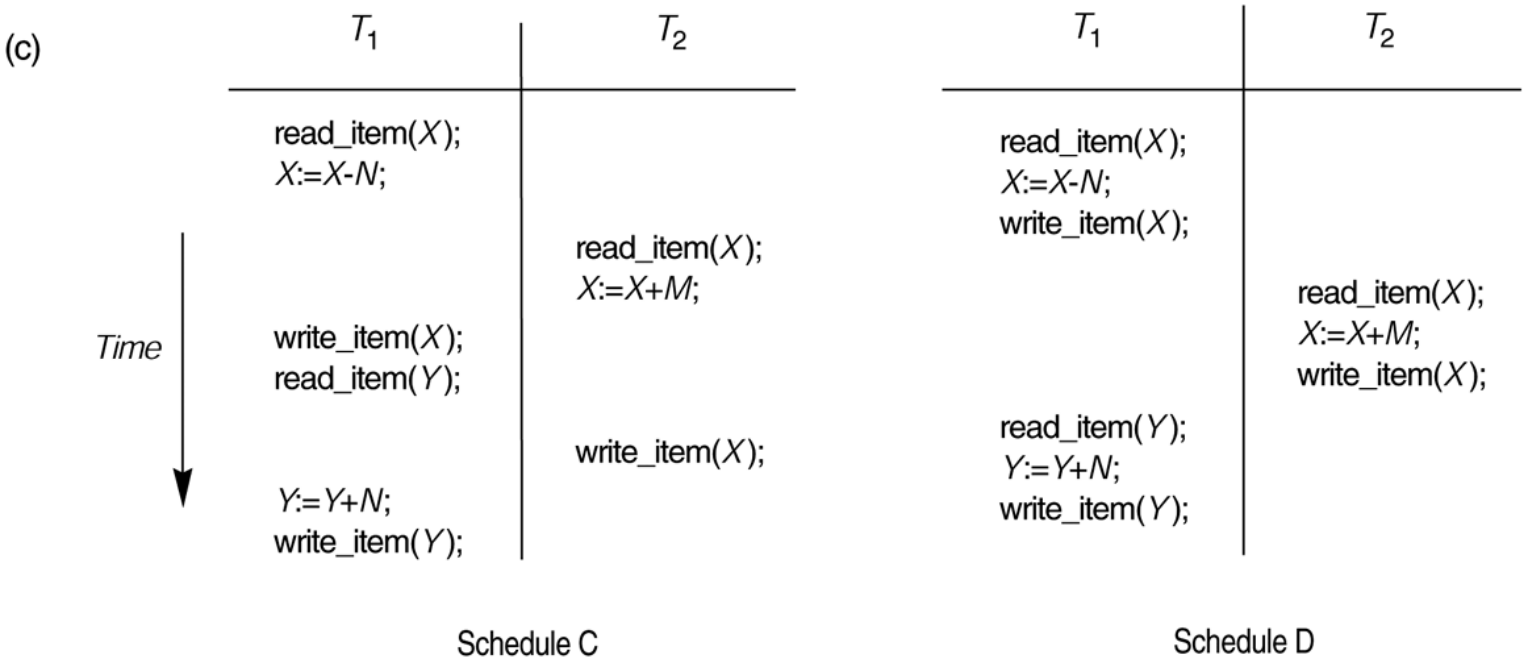
S_1	S_2
read_item(X);	read_item(X);
X:=X+10;	X:=X*1.1;
write_item(X);	write_item(X);

conflict equivalences

- 2 schedules are said to be **conflict equivalent** if the order of **any two** conflicting operations is the same in the both schedules
  - recall that 2 operations in a schedule are said to conflict if they belong to different transactions, access the same database item, and at least one of them is a write_item operation
  - example
    - S_1: $r_1(X)$; $w_2(X)$ and S_2: $w_2(X)$; $r_1(X)$ are not conflict equivalent
- a schedule *S* is **conflict serializable** if it is conflict equivalent to **some** serial schedule *S'*
  - we can reorder the nonconflicting operations in *S* until we form the equivalent **serial schedule** *S'*

example of conflict serializable schedule

- schedule C
  - conflicting operations: $r_1(X) \rightarrow w_2(X)$, $r_2(X) \rightarrow w_1(X)$, $w_1(X) \rightarrow w_2(X)$ $\Rightarrow T_1 \rightarrow T_2$, $T_2 \rightarrow T_1$, $T_1 \rightarrow T_2 \Rightarrow$ not serializable
- schedule D
  - conflicting operations: $r_1(X) \rightarrow w_2(X)$, $w_1(X) \rightarrow r_2(X)$, $w_1(X) \rightarrow w_2(X)$ $\Rightarrow T_1 \rightarrow T_2$, $T_1 \rightarrow T_2$, $T_1 \rightarrow T_2 \Rightarrow$ serializable



testing for conflict serializability

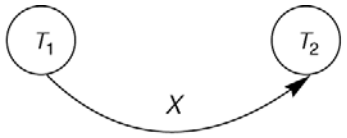
- precedence graph
  - a directed graph $G = (N, E)$ that consists of a set of nodes $N = \{T_1, ..., T_n\}$ and a set of directed edges $E = \{e_1, ..., e_m\}$
  - there is one node in the graph for each transaction T_i in the schedule
  - each edge e_i in the graph is of the form $(T_j \text{ -> } T_k)$, $1 \leq j, k \leq n$, where T_j is the starting node of e_i and T_k is the ending node of e_i
  - an edge is created if **one of the operations** in T_j appears in the schedule **before some conflicting operation** in T_k
- cycle
  - a sequence of edges $C = ((T_j \text{ -> } T_k), (T_k \text{ -> } T_p), ..., (T_i \text{ -> } T_j))$ with the property that the starting node of each edge in C (except for $T_j \text{ -> } T_k$) is the same as the ending node of the previous edge

algorithm for testing conflict serializability of S

- 1. for each transaction T_i participating in 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 -> 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 -> 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 -> T_j) in the precedence graph
- 5. schedule S is **serializable iff the precedence graph has no cycles**

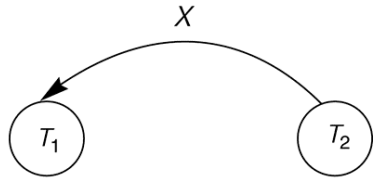
Example

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);



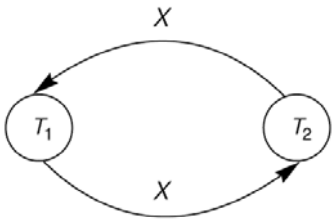
Schedule A

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);	



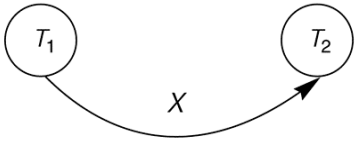
Schedule B

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;	
	read_item(X);
	X:=X+M;
write_item(X);	
read_item(Y);	
	write_item(X);
Y:=Y+N;	
write_item(Y);	



Schedule D

another example

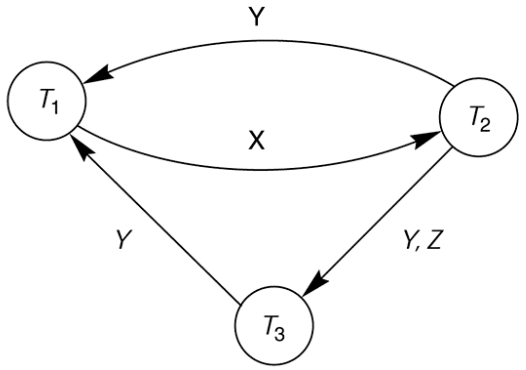
transaction T_1
read_item (X); write_item (X); read_item (Y); write_item (Y);

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);

another example (cont.)

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);	



Equivalent serial schedules

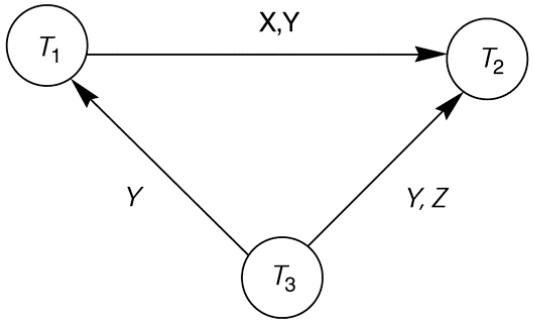
None

Reason

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

Schedule E

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);	

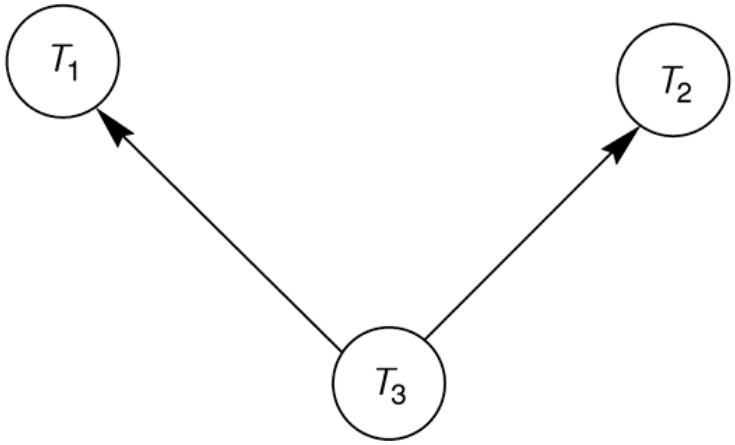


Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

Schedule F

a precedence graph with two equivalent serial schedules



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

$T_3 \rightarrow T_2 \rightarrow T_1$

uses of serializability

- a serializable schedule gives the benefits of concurrent execution while being able to be correct
- if the transactions are executed at will and then the resulting schedule is tested for serializability, we must cancel the effect of the schedule if it turns out not to be serializable
- hence, the approach taken in most commercial DBMS is to design concurrency control protocols that will ensure serializability of all schedules in which the transactions participate
  - two-phase locking
  - timestamp ordering
  - multiversion protocols
  - optimistic protocols

other types of equivalence of schedules

- serializability of schedules is sometimes considered to be too restrictive
- example: debit-credit transactions
  - T_1: $r_1(X)$; $X = X - 10$; $w_1(X)$; $r_1(Y)$; $Y = Y + 10$; $w_1(Y)$;
  - T_2: $r_2(Y)$; $Y = Y - 20$; $w_2(Y)$; $r_2(X)$; $X = X + 20$; $w_2(X)$;
  - S_h: $r_1(X)$; $w_1(X)$; $r_2(Y)$; $w_2(Y)$; $r_1(Y)$; $w_1(Y)$; $r_2(X)$; $w_2(X)$; => nonserializable, but still correct

transaction support in SQL

- no explicit Begin_Transaction statement
- every transaction must have an explicit end statement, which is either a COMMIT or a ROLLBACK
- each transaction has certain characteristics
  - access mode: READ ONLY or READ WRITE
  - diagnostic area size: the # of conditions that can be held simultaneously
  - isolation level: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, or SERIALIZABLE

Isolation level	Dirty read	Nonrepeatable read	Phantom
READ UNCOMMITTED	yes	yes	yes
READ COMMITTED	no	yes	yes
REPEATABLE READ	no	no	yes
SERIALIZABLE	no	no	no

types of violations

- dirty read
  - a transaction T_1 may read the update of a transaction T_2, which has not yet committed. If T_2 fails and is aborted, then T_1 would have read a value that does not exist and is incorrect
- nonrepeatable read
  - a transaction T_1 may read a given value from a table. If another transaction T_2 later updates that value and T_1 reads that value again, T_1 will see a different value
- phantoms
  - a transaction T_1 may read a set of rows from a table based on some condition specified in the SQL WHERE clause. Now T_2 inserts a new row that also satisfies the condition used in T_1, into the table used by T_1. if T_1 is repeated, then T_1 will see a phantom, a row that previously did not exist

example SQL code

```
EXEC SQL WHENEVER SQLERROR GOTO UNDO;
EXEC SQL SET TRANSACTION
        READ WRITE
        DIAGNOSTIC SIZE 5
        ISOLATION LEVEL SERIALIZABLE;
EXEC SQL INSERT INTO EMPLOYEE (FNAME, LNAME, SSN, DNO, SAL)
        VALUES ('J', 'HOWE', '99999999', 1, 800000);
EXEC SQL UPDATE EMPLOYEE
        SET SAL = SAL * 2.0 WHERE DNO = 1;
EXEC SQL COMMIT;
GOTO THE_END;
UNDO: EXEC SQL ROLLBACK;
THE_END; ...;
```