Transaction Processing Concepts and Theory

Truong Tuan Anh
CSE-HCMUT

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

Transaction Processing

- **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.

Transaction Processing

- **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.

Transaction Processing

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

- **A database -** collection of named data items
- **Granularity of data** - a field, a record , 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*.

Transaction Processing

READ AND WRITE OPERATIONS:

- Basic unit of data transfer from the disk to the computer main memory is **one block**.
- Data item (what is read or written):
  - the field of some records in the database
  - 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*.

Transaction Processing

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).

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);

Transaction Processing

Why Concurrency Control is needed:

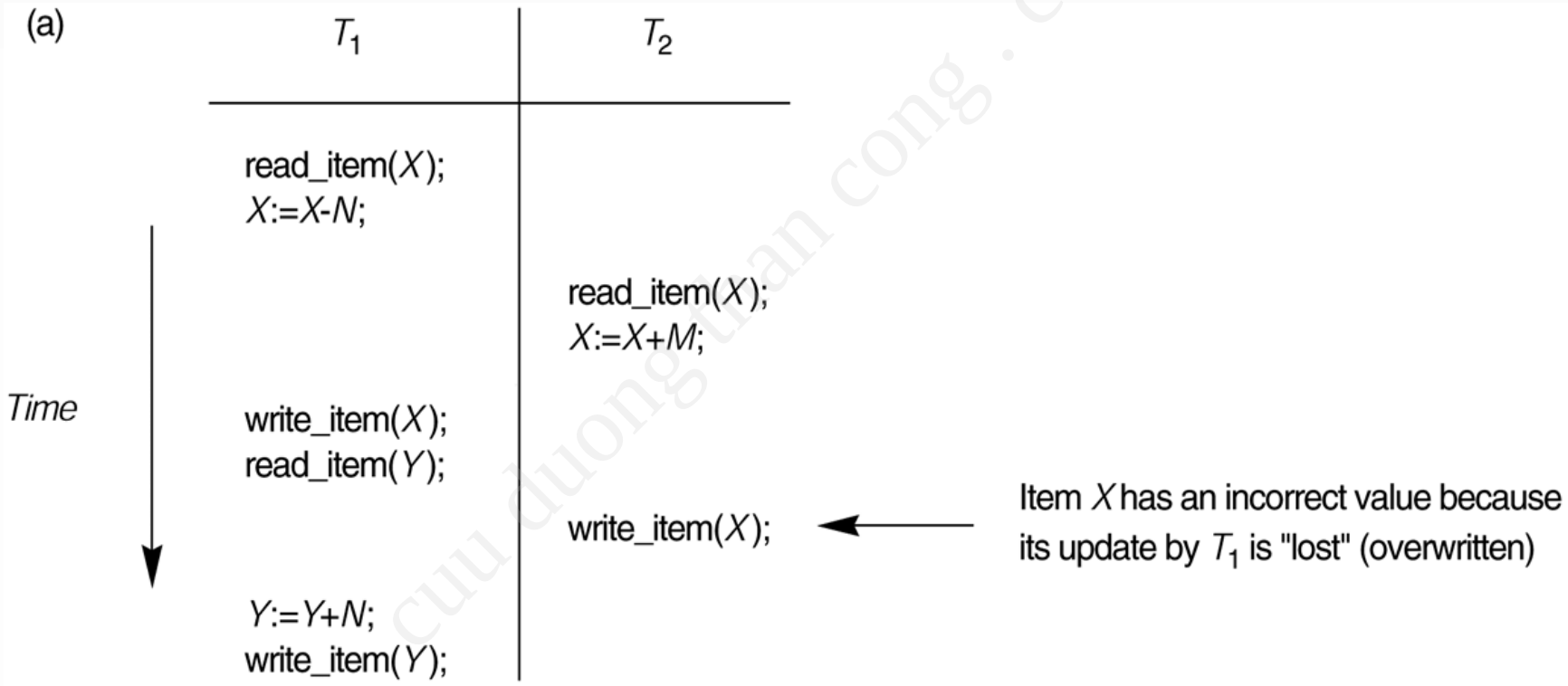
- **The Lost Update Problem.**

  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.

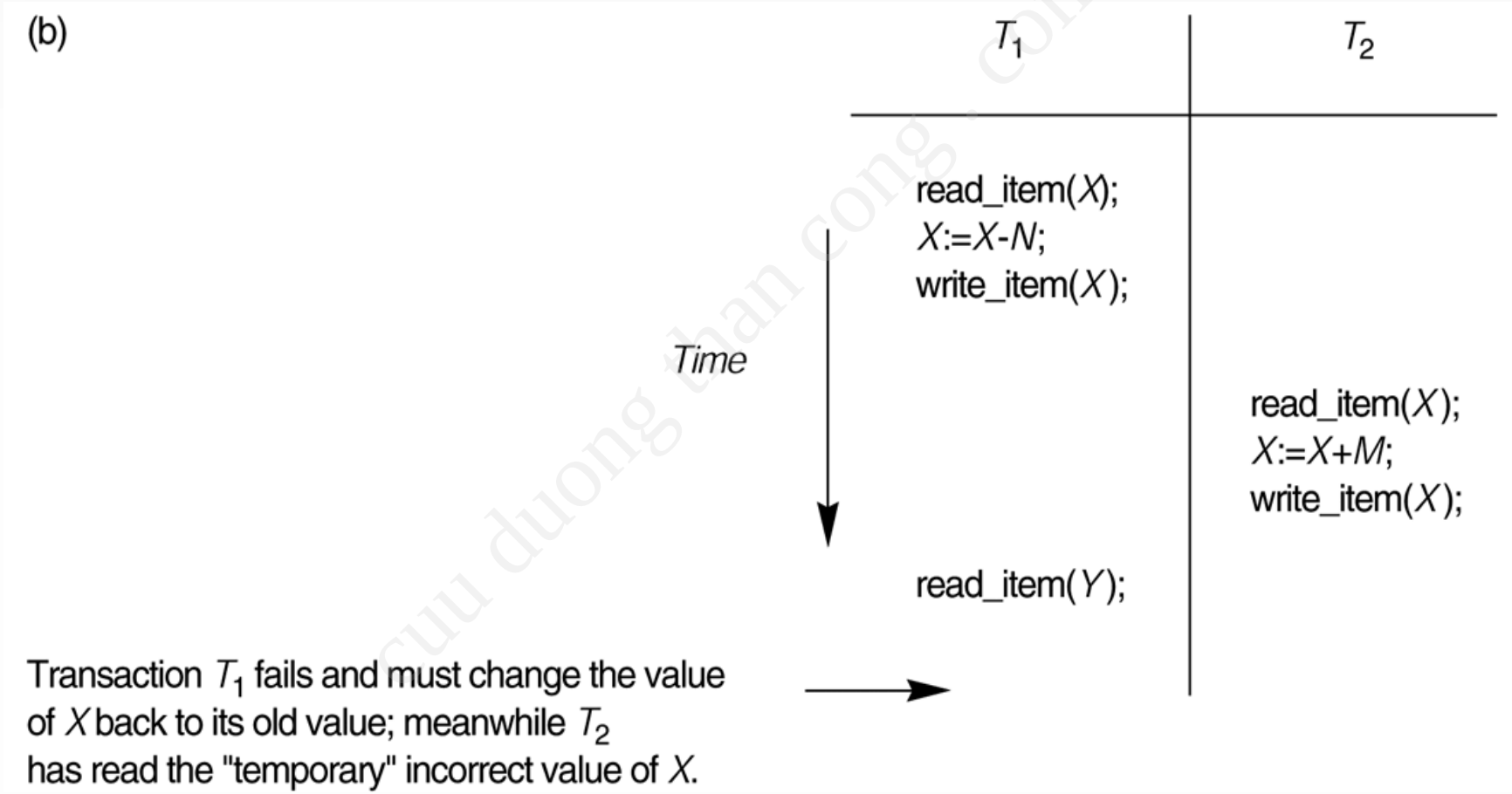
- **The Temporary Update (or Dirty Read) Problem.**

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

The Lost Update Problem



The Temporary Update Problem



Transaction Processing (cont.)

Why Concurrency Control is needed (cont.):

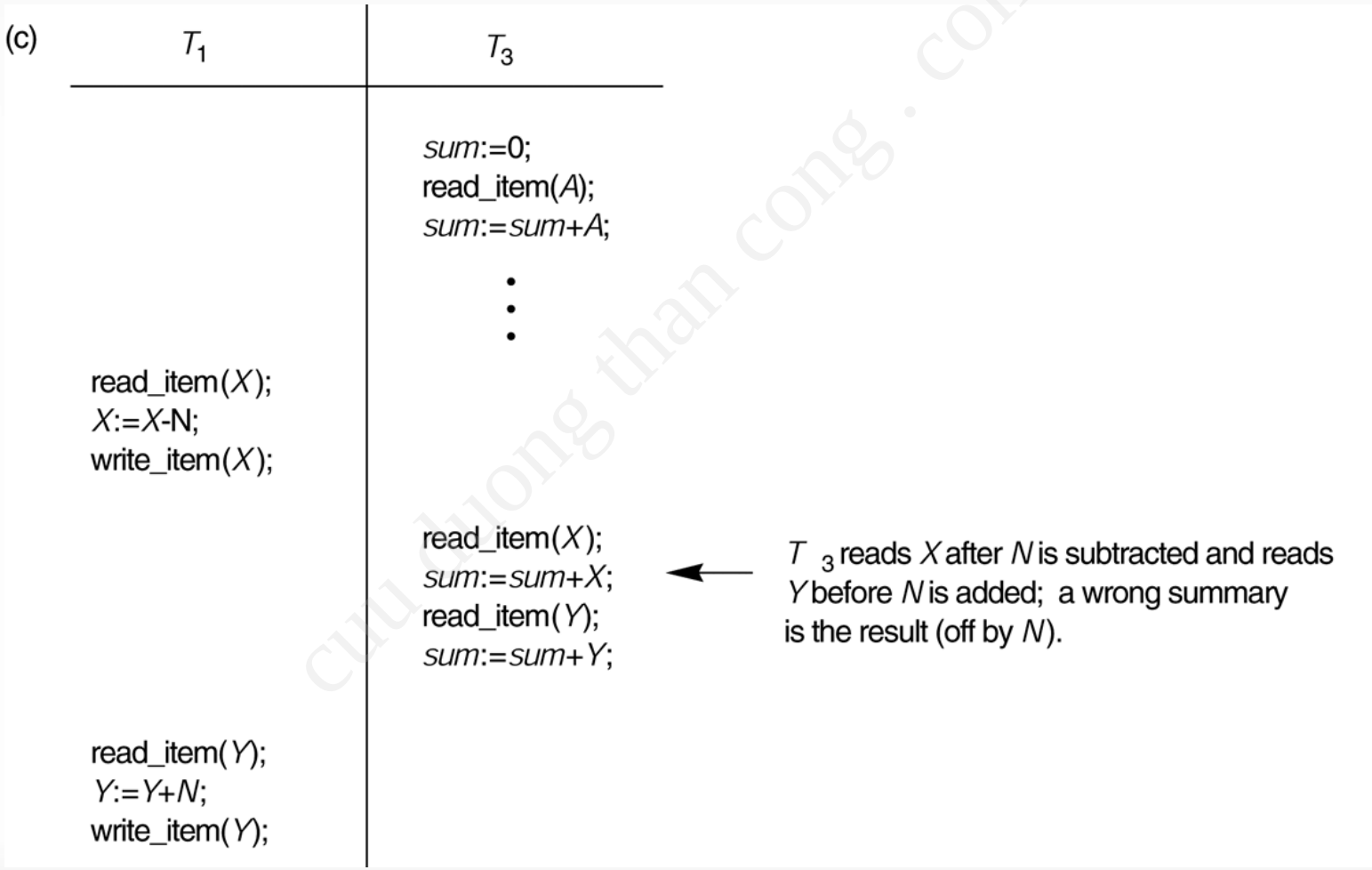
- **The Incorrect Summary Problem .**

  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.

- **The Unrepeatable Read Problem:**

  Transaction *T* reads the same item twice and the item is changed by another transaction *T'* between the two reads.

The Incorrect Summary Problem



Transaction Processing

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 operations 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.

Transaction Processing

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.

Transaction Processing

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.

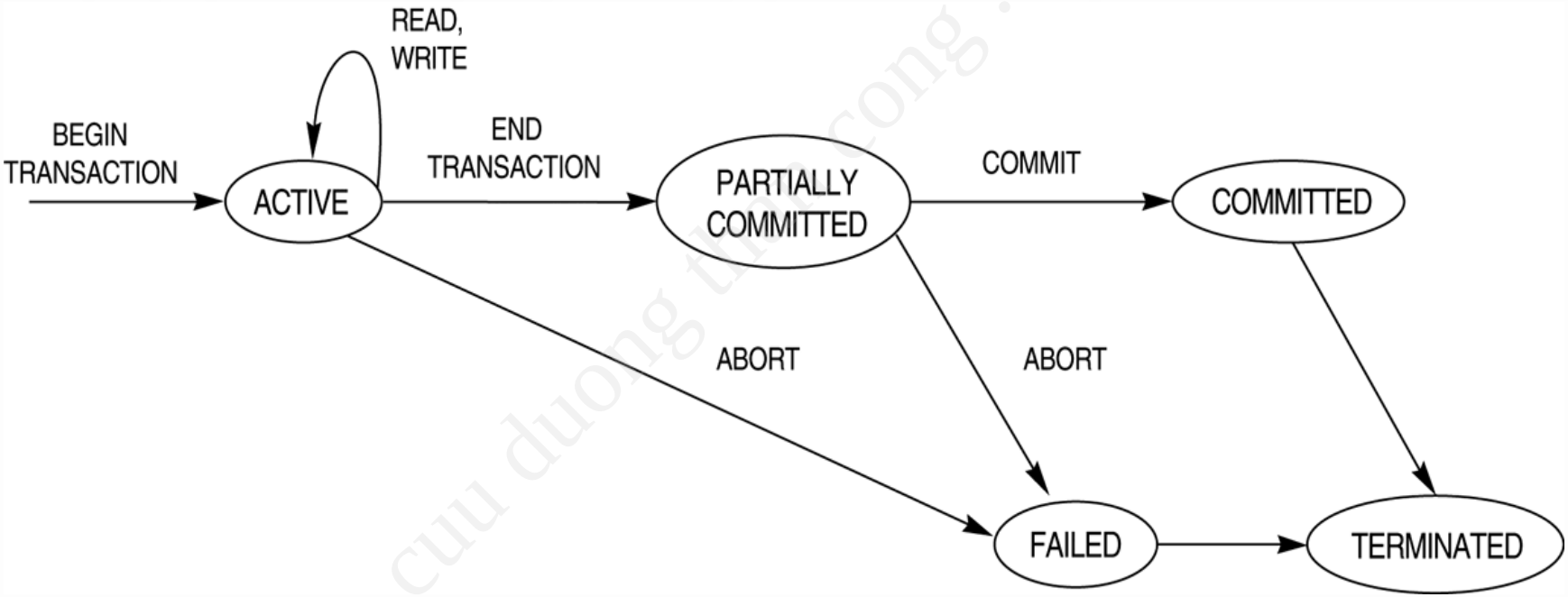
Transaction and System Concepts

- 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

State Transition Diagram

Illustrate the states for transaction execution



Transaction and System Concepts

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
- **end_transaction:**
  - This specifies that read and write transaction operations have ended and marks the end limit of transaction execution.
  - may be necessary to **check** whether the changes introduced by the transaction can 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

Recovery manager keeps track of the following operations

- **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

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.

Transaction and System Concepts

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 → 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.

Transaction and System Concepts

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

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*.

Transaction and System Concepts

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 recovery techniques.
- 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.*
- 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

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.

Desirable Properties of Transactions

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

Properties (cont.):

- **Isolation**: A transaction should appear as though it is being executed in isolation from other transaction. That is, the execution of a transaction should not be interfered with by any other transaction executing concurrently.

- **Durability or permanency**: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

Characterizing Schedules based on Recoverability

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

- A **schedule** (or **history**) S of n transactions T_1, T_2, ..., T_n :
  - Constraint: for each transaction T_i that participates in S, the operations of T_1 in S **must appear in the same order** in which they occur in T_1.
  - However, that operations from other transactions T_j can be interleaved with the operations of T_i in S.

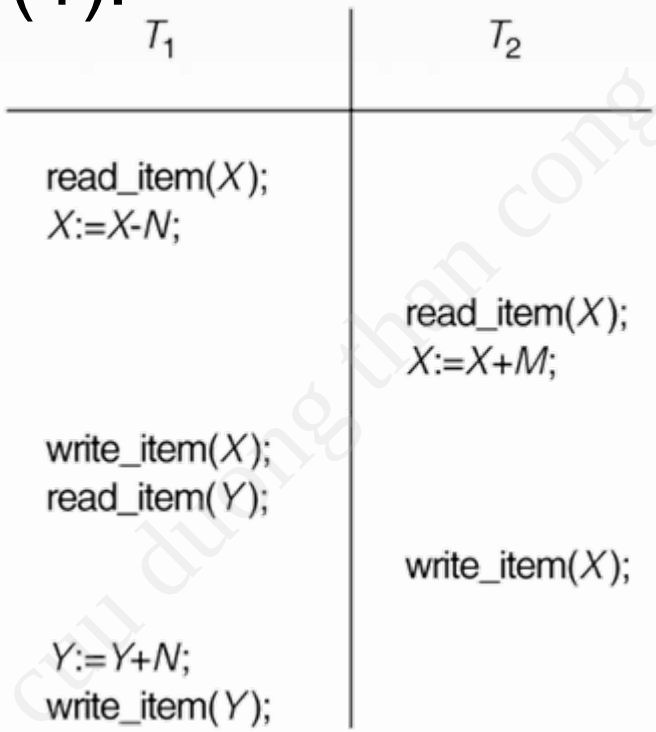
Characterizing Schedules based on Recoverability

- Notations:

Notation	Description
$r_i(X)$	read_item(X) - transaction T_i
$w_i(X)$	write_item(X) - transaction T_i
c_i	commit - transaction T_i
a_i	abort - transaction T_i

Characterizing Schedules based on Recoverability

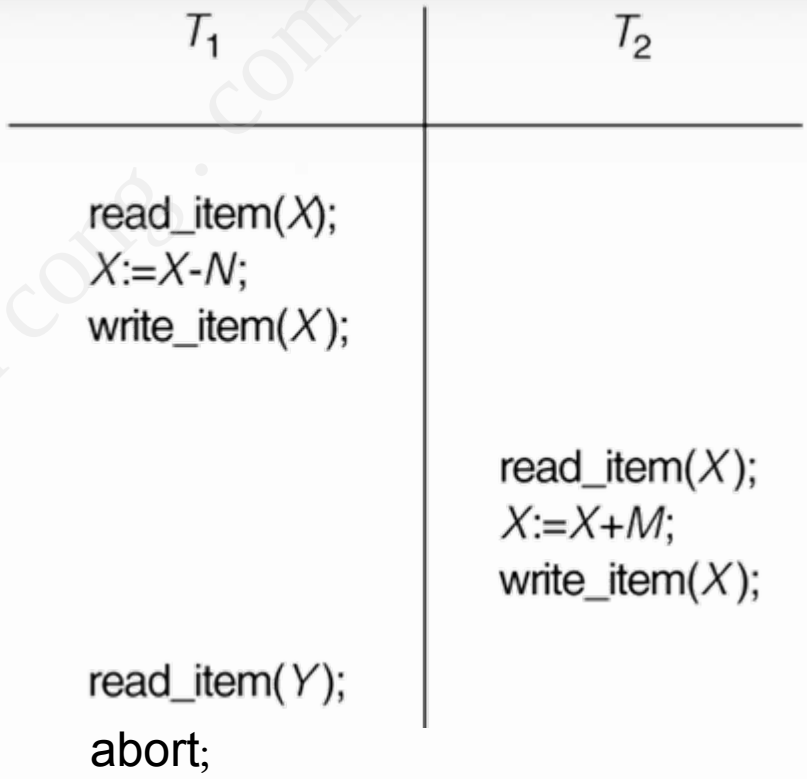
- Example (1):



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

Characterizing Schedules based on Recoverability

- Example (2):



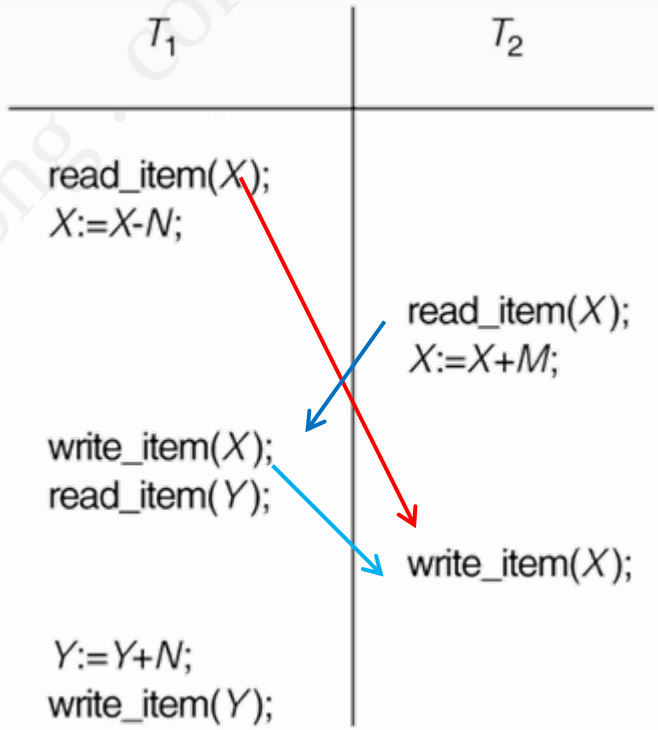
- S_b: $r_1(X)$; $w_1(X)$; $r_2(X)$; $w_2(X)$; $r_1(Y)$; a_1;

Characterizing Schedules based on Recoverability

- Two operations in a schedule are said to **conflict** if they satisfy **all** followings:
  - (1) they belong to *different transactions*.
  - (2) they access the *same item* X.
  - (3) *at least one* of the operation is a write_item(X)

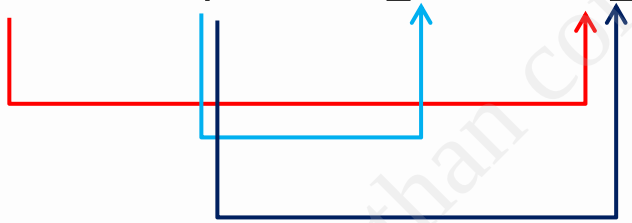
Characterizing Schedules based on Recoverability

- Example (1):
  - conflict:
    - $r_1(X)$ and $w_2(X)$
    - $r_2(X)$ and $w_1(X)$
    - $w_1(X)$ and $w_2(X)$
  - Not conflict:
    - $r_1(X)$ and $r_2(X)$
    - $r_1(Y)$ and $w_2(X)$
    - $r_1(X)$ and $w_1(X)$
    - …



Characterizing Schedules based on Recoverability

- Example (2):
  - S_b: $r_1(X)$; $w_1(X)$; $r_2(X)$; $w_2(X)$; $r_1(Y)$; a_1;



  - Conflict:
    - $r_1(X)$ and $w_2(X)$
    - $w_1(X)$ and $r_2(X)$
    - $w_1(X)$ and $w_2(X)$

Characterizing Schedules based on Recoverability

Schedules classified on recoverability:

- **Recoverable schedule:** 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.

- **Cascadeless schedule:** A schedule which every transaction reads only the items that are written by committed transactions.

  Schedules requiring cascaded rollback: A schedule in which uncommitted transactions that read an item from a failed transaction must be rolled back.

Characterizing Schedules based on Recoverability

Schedules classified on recoverability (cont.):

- **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.

Characterizing Schedules based on Recoverability

- Example of **Recoverable schedule** :

  S_a': $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; c_2; $w_1(Y)$; c_1;

  - Lost update

- Example of **Not Recoverable schedule** :

  S_c: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; c_2; a_1;

Characterizing Schedules based on Recoverability

- Example of **Cascading Rollback**:

  S_d: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; c_1; c_2;

  - S_d : **Recoverable schedule**

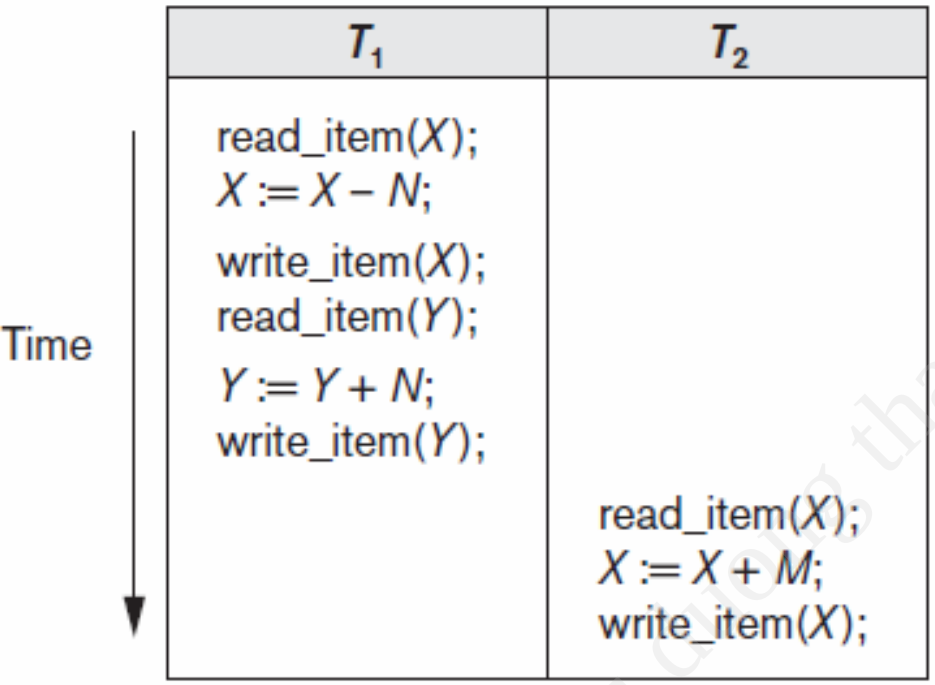
- Example of **Cascadeless schedule**:

  S'_d: $r_1(X)$; $w_1(X)$; $r_1(Y)$; $w_1(Y)$; c_1; $r_2(X)$; $w_2(X)$; c_2;

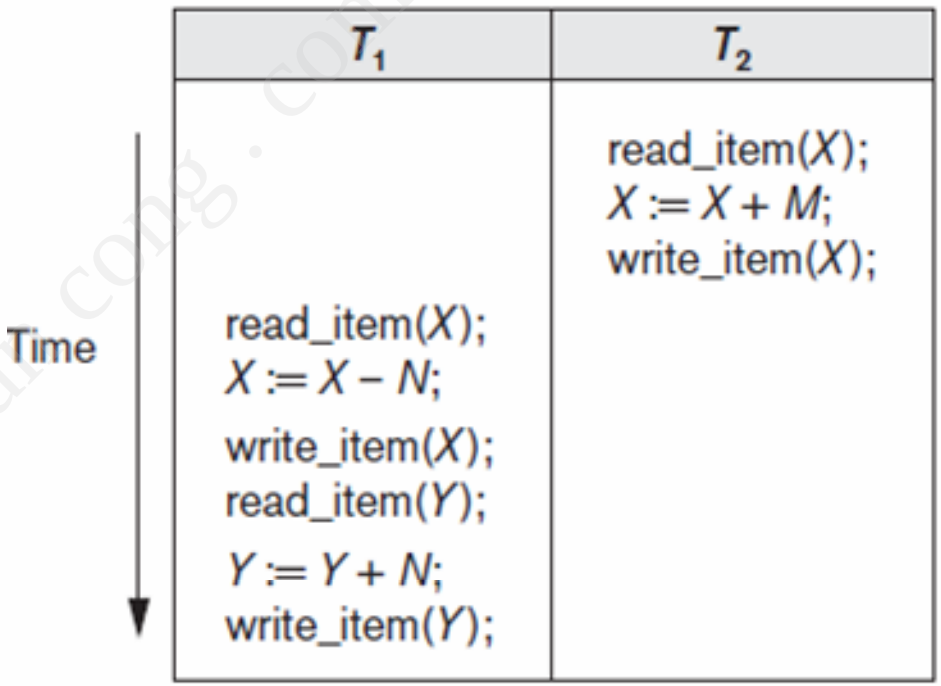
Characterizing Schedules based on Serializability

- **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. 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.

Characterizing Schedules based on Serializability



Schedule A

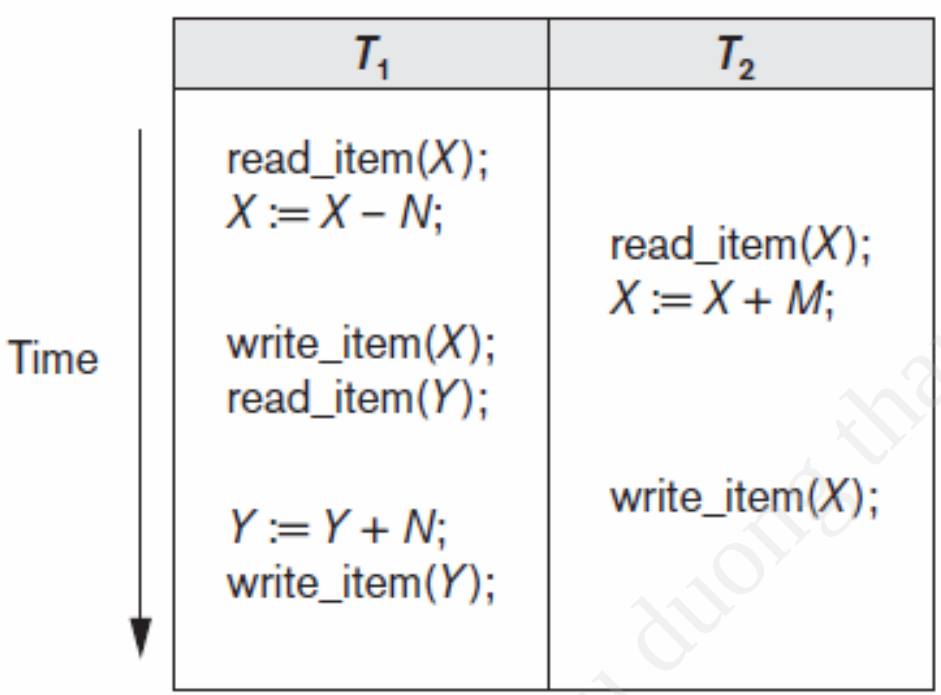


Schedule B

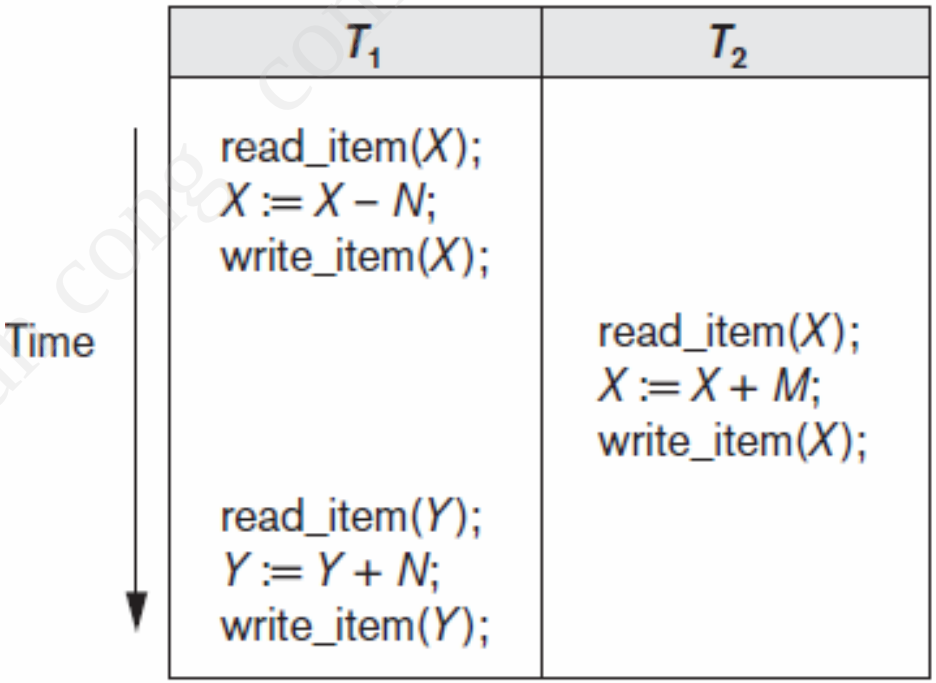
Serial Schedules:

(A) T_1 followed by T_2 (B) T_2 followed by T_1

Characterizing Schedules based on Serializability



Schedule C



Schedule D

Two nonserial Schedules

Characterizing Schedules based on Serializability

- **Result equivalent**: Two schedules are called *result equivalent* if they produce the same final state of the database.
- **Conflict equivalent**: Two schedules are said to be *conflict equivalent* if the order of any two *conflicting operations* is the same in both schedules.
  - Two operations in a schedule are said to *conflict* if they belong to different transactions, access the same data item, and at least one of the two operations is a *write_item* operation.

Characterizing Schedules based on Serializability

- **Conflict serializable**: A schedule *S* is said to be *conflict serializable* if it is *conflict equivalent* to some serial schedule *S'*.
  - In such a case, we can reorder the *nonconflicting* operations in S until we form the equivalent serial schedule S'.

Exercise

- Given two following transactions:

T1: r1(A); w1(A); r1(B);w1(B);

T2: r2(A); w2(A); r2(B);w2(B);

Prove that the schedule:

- S: r1(A);w1(A); r2(A); w2(A); r1(B);w1(B); r2(B);w2(B); is *conflict-serializable*.

Characterizing Schedules based on Serializability

- Being serializable is not the same as being serial

- 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.

Characterizing Schedules based on Serializability

- Serializability is hard to check.
  - Interleaving of operations occurs in an operating system through some scheduler
  - Difficult to determine beforehand how the operations in a schedule will be interleaved.

Characterizing Schedules based on Serializability

Practical approach:

- Come up with methods (protocols) to ensure serializability.
- It's not possible to determine when a schedule begins and when it ends.

→ Hence, we reduce the problem of checking the whole schedule to checking only a *committed project* of the schedule (i.e. operations from only the committed transactions)

- Current approach used in most DBMSs:
  - Use of locks with two phase locking

Characterizing Schedules based on Serializability

Testing for conflict serializability

- Precedence graph (serialization graph) $G = (N, E)$
  - Directed graph
  - Set of Nodes: $N = \{T_1, T_2, \ldots, T_n\}$
  - Directed edge: $E = \{e_1, e_2, \ldots, e_m\}$
    - e_i: $T_j \rightarrow T_k$ if one of the operations in T_j appears in the schedule before *some* *conflicting operation* in T_k

Characterizing Schedules based on Serializability

Testing for conflict serializability (cont.)

- **Algorithm**
  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.

Characterizing Schedules based on Serializability

Testing for conflict serializability (cont.)

- **Algorithm (cont.)**

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**.

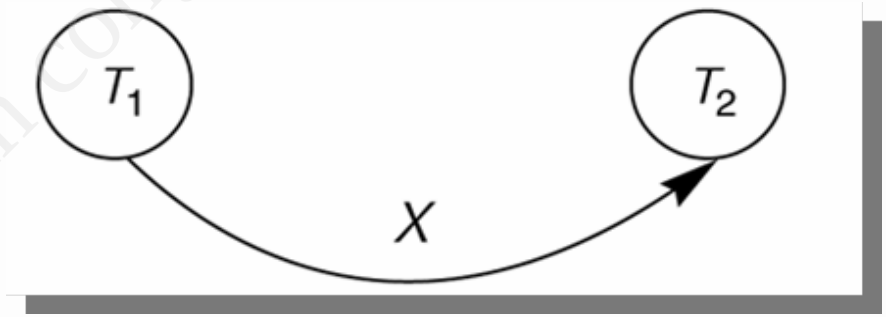
Example of Serializability Testing

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);

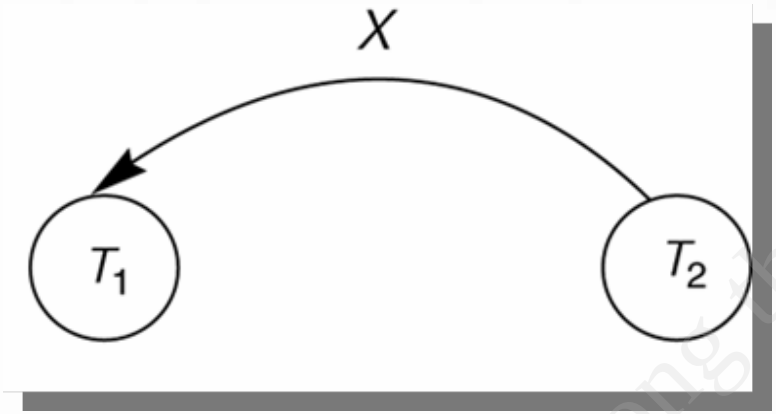
Time

Schedule A



Serializable schedule

Example of Serializability Testing



Serializable schedule

Time

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);	

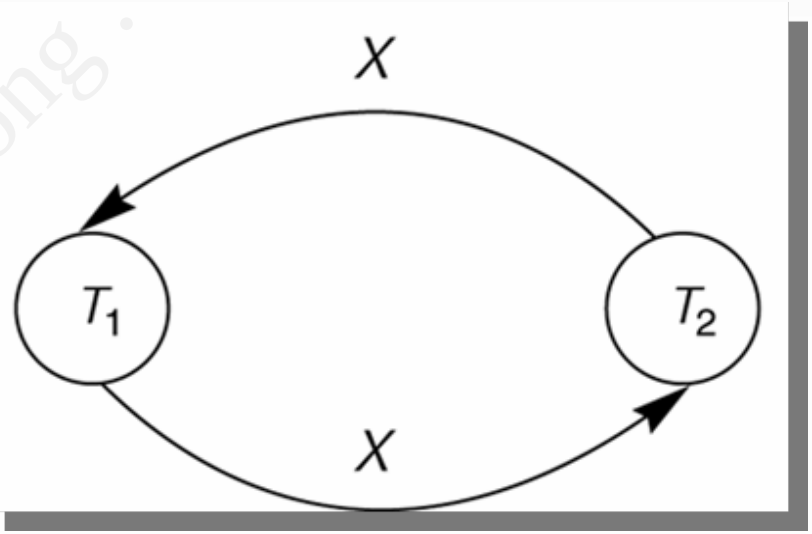
Schedule B

Example of Serializability Testing

Time

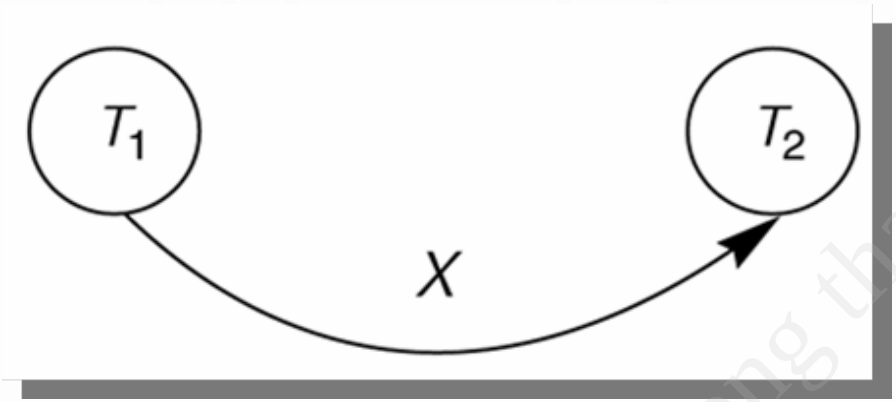
T_1	T_2
read_item(X);	
$X := X - N$;	
	read_item(X);
	$X := X + M$;
write_item(X);	
read_item(Y);	
	write_item(X);
$Y := Y + N$;	
write_item(Y);	

Schedule C



Not Serializable schedule

Example of Serializability Testing



Time

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

Serializable schedule

Example: (a) The READ and WRITE Operations of Three Transactions T_1, T_2, and T_3.

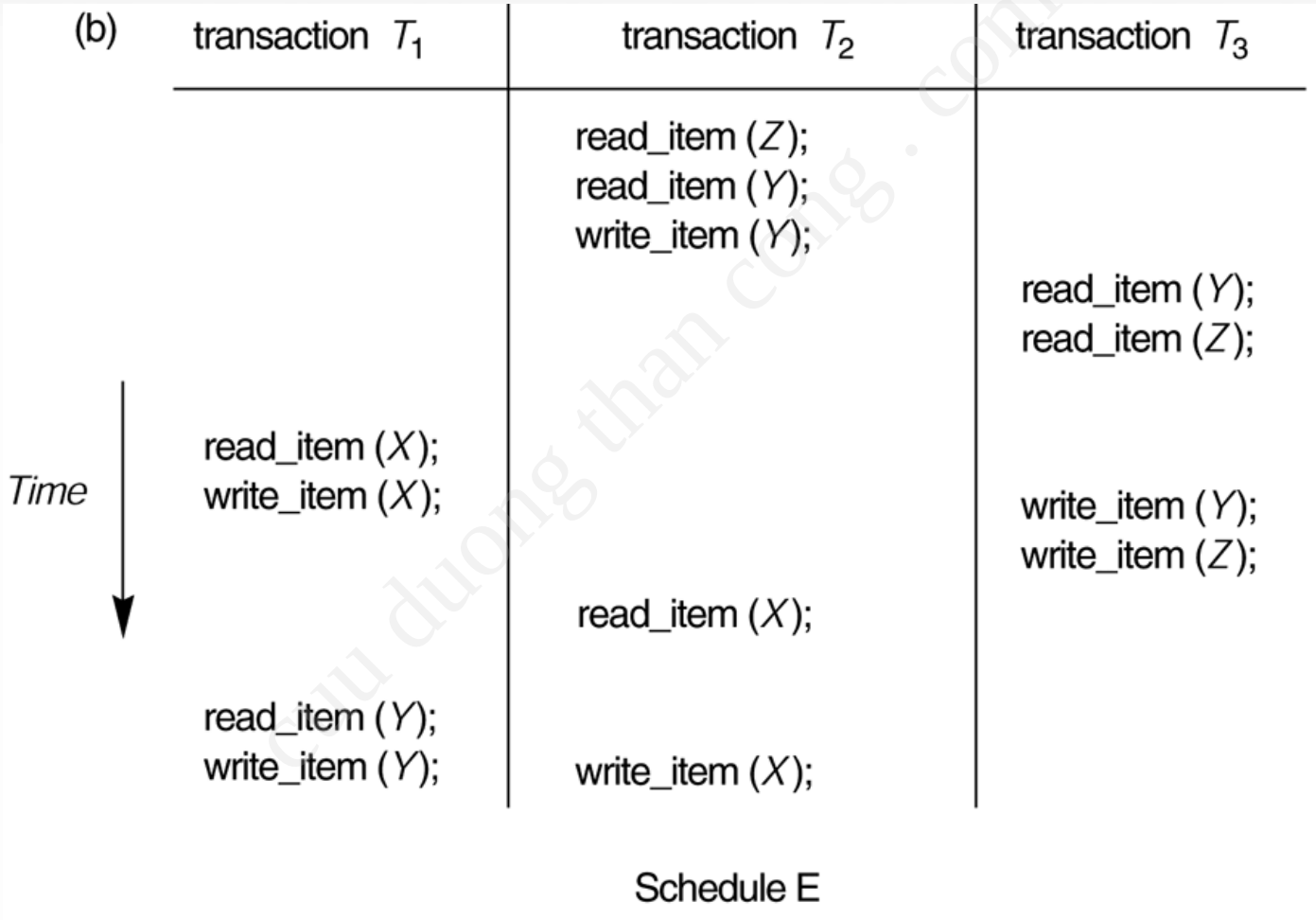
(a)

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

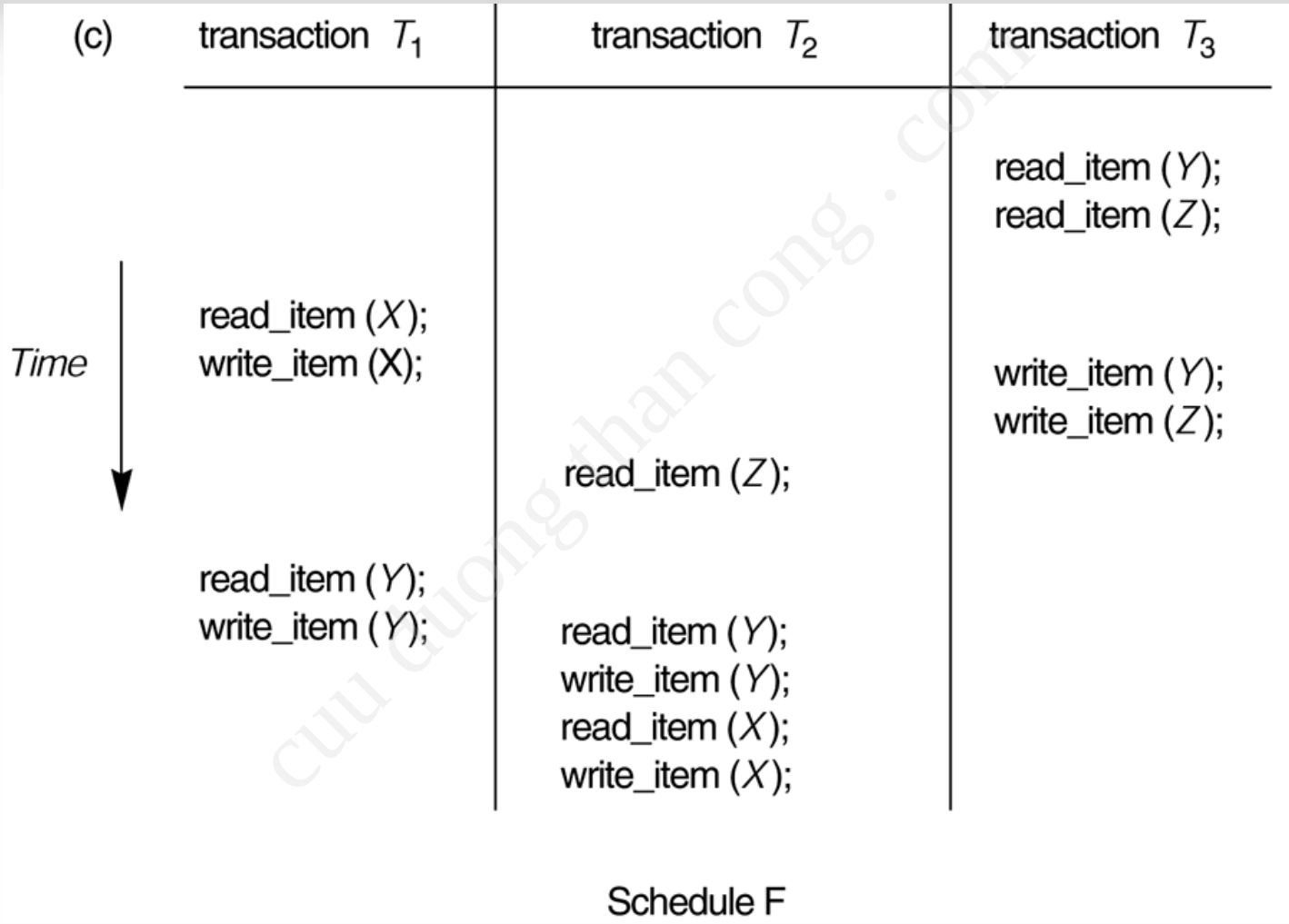
transaction T_2
read_item (Z); read_item (Y); write_item (Y); read_item (X); write_item (X);

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

Example: (b) Schedule *E*

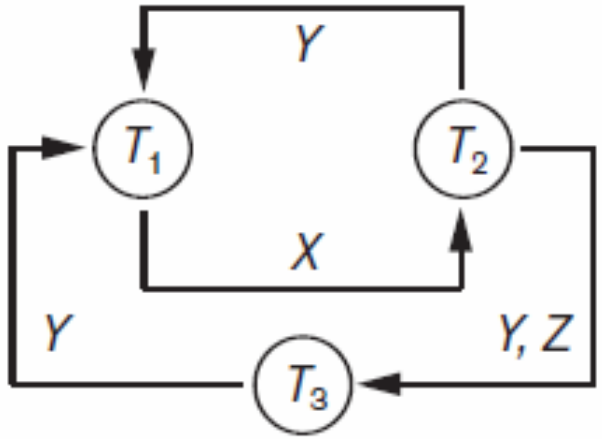


Example: (c) Schedule *F*



Schedule F

(d)



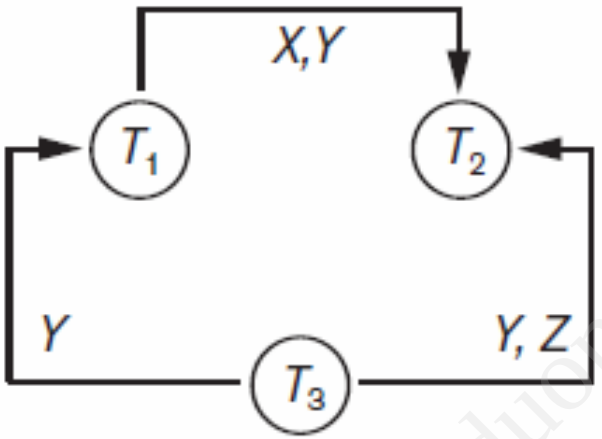
Equivalent serial schedules

None

Reason

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

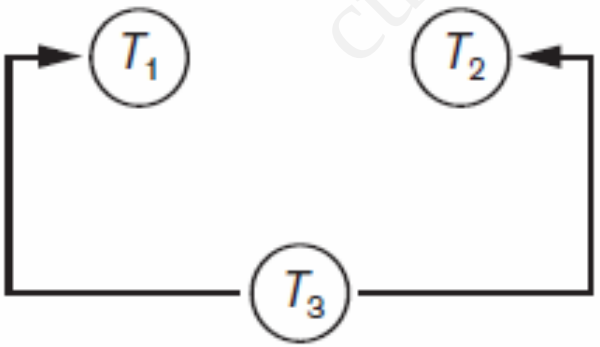
(e)



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

(f)



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

$T_3 \rightarrow T_2 \rightarrow T_1$

Exercises

- Which of the following schedules is (conflict) serializable? For each serializable schedule, determine the equivalent serial schedules.

a. r1(X); r3(X); w1(X); r2(X); w3(X);

b. r1(X); r3(X); w3(X); w1(X); r2(X);

c. r3(X); r2(X); w3(X); r1(X); w1(X);

d. r3(X); r2(X); r1(X); w3(X); w1(X);

Exercises

- Consider the three transactions T1, T2, and T3, and the schedules S1 and S2 given below. Draw the serializability (precedence) graphs for S1 and S2, and state whether each schedule is serializable or not. If a schedule is serializable, write down the equivalent serial schedule(s).

T1: r1 (X); r1 (Z); w1 (X);
T2: r2 (Z); r2 (Y); w2 (Z); w2 (Y);
T3: r3 (X); r3 (Y); w3 (Y);

S1: r1 (X); r2 (Z); r1 (Z); r3 (X); r3 (Y); w1 (X); w3 (Y); r2 (Y); w2 (Z); w2 (Y);
S2: r1 (X); r2 (Z); r3 (X); r1 (Z); r2 (Y); r3 (Y); w1 (X); w2 (Z); w3 (Y); w2 (Y);

Transaction Support in SQL2

- A single SQL statement is *always considered to be atomic*. Either the statement completes execution without error or it fails and leaves the database unchanged.
- With SQL, there is no explicit Begin Transaction statement. Transaction initiation is done implicitly when particular SQL statements are encountered.
- Every transaction must have an explicit end statement, which is either a COMMIT or ROLLBACK.

Transaction Support in SQL2

Characteristics specified by a SET TRANSACTION statement in SQL2:

- **Access mode:** READ ONLY or READ WRITE. The default is READ WRITE unless the isolation level of READ UNCOMITTED is specified, in which case READ ONLY is assumed.
- **Diagnostic size** n, specifies an integer value n, indicating the number of conditions that can be held simultaneously in the diagnostic area. (Supply user feedback information)

Transaction Support in SQL2

Characteristics specified by a SET TRANSACTION statement in SQL2 (cont.):

- **Isolation level** <isolation>, where <isolation> can be READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ or SERIALIZABLE. The default is SERIALIZABLE.

  With SERIALIZABLE: the interleaved execution of transactions will adhere to our notion of serializability. However, if any transaction executes at a lower level, then serializability may be violated.

Transaction Support in SQL2

Potential problem with lower isolation levels:

- **Dirty Read**: Reading a value that was written by a transaction which failed.
- **Nonrepeatable Read**: Allowing another transaction to write a new value between multiple reads of one transaction.

  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. Consider that T_1 reads the employee salary for Smith. Next, T_2 updates the salary for Smith. If T_1 reads Smith's salary again, then it will see a different value for Smith's salary.

Transaction Support in SQL2

Potential problem with lower isolation levels (cont.):

- **Phantoms**: New rows being read using the same read with a condition.
  - A transaction T_1 may read a set of rows from a table, perhaps based on some condition specified in the SQL WHERE clause.
  - A transaction T_2 inserts a new row that also satisfies the WHERE clause condition of T_1, into the table used by T_1.
  - If T_1 is repeated, then T_1 will see a row that previously did not exist, called a **phantom**.

Transaction Support in SQL2

Sample SQL transaction:

```
EXEC SQL whenever sqlerror go to UNDO;
EXEC SQL SET TRANSACTION
          READ WRITE
          DIAGNOSTICS SIZE 5
          ISOLATION LEVEL SERIALIZABLE;
EXEC SQL INSERT
          INTO EMPLOYEE (FNAME, LNAME, SSN, DNO, SALARY)
          VALUES ('Robert','Smith','991004321',2,35000);
EXEC SQL UPDATE EMPLOYEE
          SET SALARY = SALARY * 1.1
          WHERE DNO = 2;
EXEC SQL COMMIT;
          GOTO  THE_END;
UNDO: EXEC SQL ROLLBACK;
THE_END:  ...
```

Transaction Support in SQL2

Possible violation of serializabilty:

	Type of Violation		
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