Transaction Processing: Theory and Concept

Swarup Kr Ghosh
swarupg1@gmail.com

Introduction

- Transaction Processing systems are system with large database and hundred of users that are executing database transaction
- Require high availability and fast response time for hundreds of concurrent users

Examples

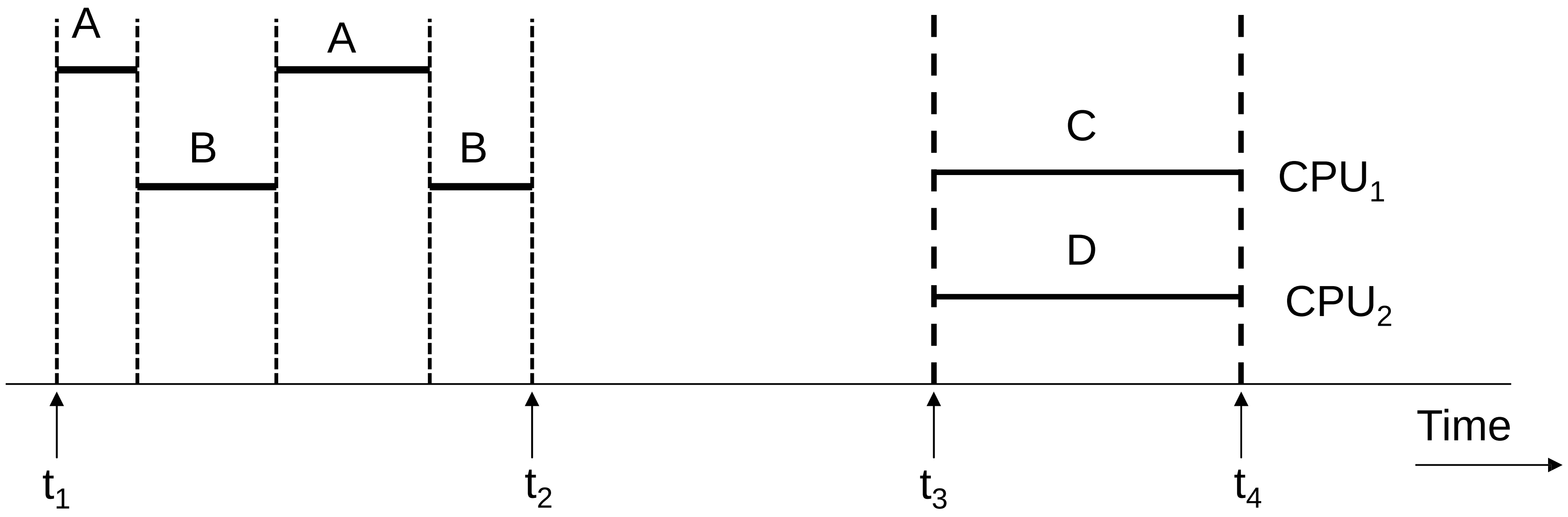
- Airline/ Railway Reservation Systems
- Banking Systems (ATM, ...)
- Trading and Brokerage Systems
- Hotel / Hospital Systems
- Standard Commercial Systems

Single User Versus Multi Users System

- **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

Single User Versus Multi User System

- Interleaved Process
- Parallel Process



Transaction, Read and Write Operations

- A transaction is an executing program that forms a *logical unit* of database processing
- Transaction includes one or more database access operation
  - Insertion
  - Deletion
  - Modification
  - Retrieval
- Database operation that form a transaction can be embedded within an application program or can be specified interactively via high level query language like SQL

Transaction, Read and Write Operations

- **Transaction boundaries**: Begin and End transaction
- An **application program** may contain several transactions separated by the Begin and End transaction boundaries
- Read-only Transaction
- A database is specified as a collection of named data items
- The size of a data item is called its granularity

Read and Write Operations

- Basic database access operations include
  - **read_item (x) :** Reads a database item named x into a program variable
  - **write_item (x) :** Writes the value of program variable x into the database item named x
- Basic unit of data transfer from the disk to the computer main memory is one block
- In general, a data item (what is read or written) will be the field of some record in the database, although it may be a larger unit such as a record or even a whole block

Read and Write Operations

- **read_item (x) includes the following steps:**

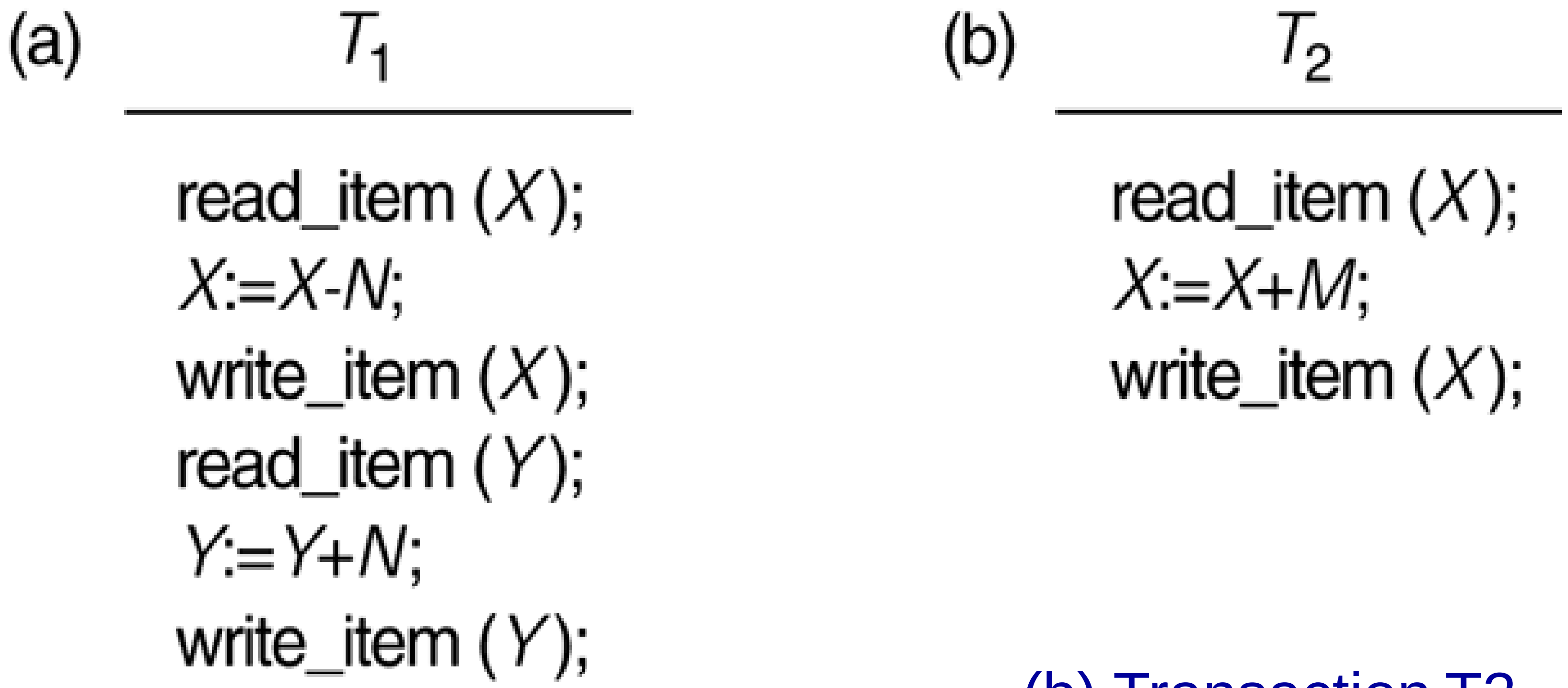
1. Find the address of the disk block that contains item x
2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer)
3. Copy item x from the buffer to the program variable named x

Read and Write Operations

- **write_item (x) 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) 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);
```

(b) Transaction T2

(a) Transaction T1

Example of Fund Transfer

- Transaction to transfer $50 from account A to account B:

1. **read**(A)
2. $A := A - 50$
3. **write**(A)
4. **read**(B)
5. $B := B + 50$
6. **write**(B)

- if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result
- the sum of A and B should be unchanged by the execution of the transaction

Example of Fund Transfer (Contd.)

- If between steps 3 and 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database (the sum *A* + *B* will be less than it should be)
  - One solution, running transactions **serially**, that is one after the other
  - However, executing multiple transactions concurrently has significant benefits, as we will see later
- Once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist despite failures

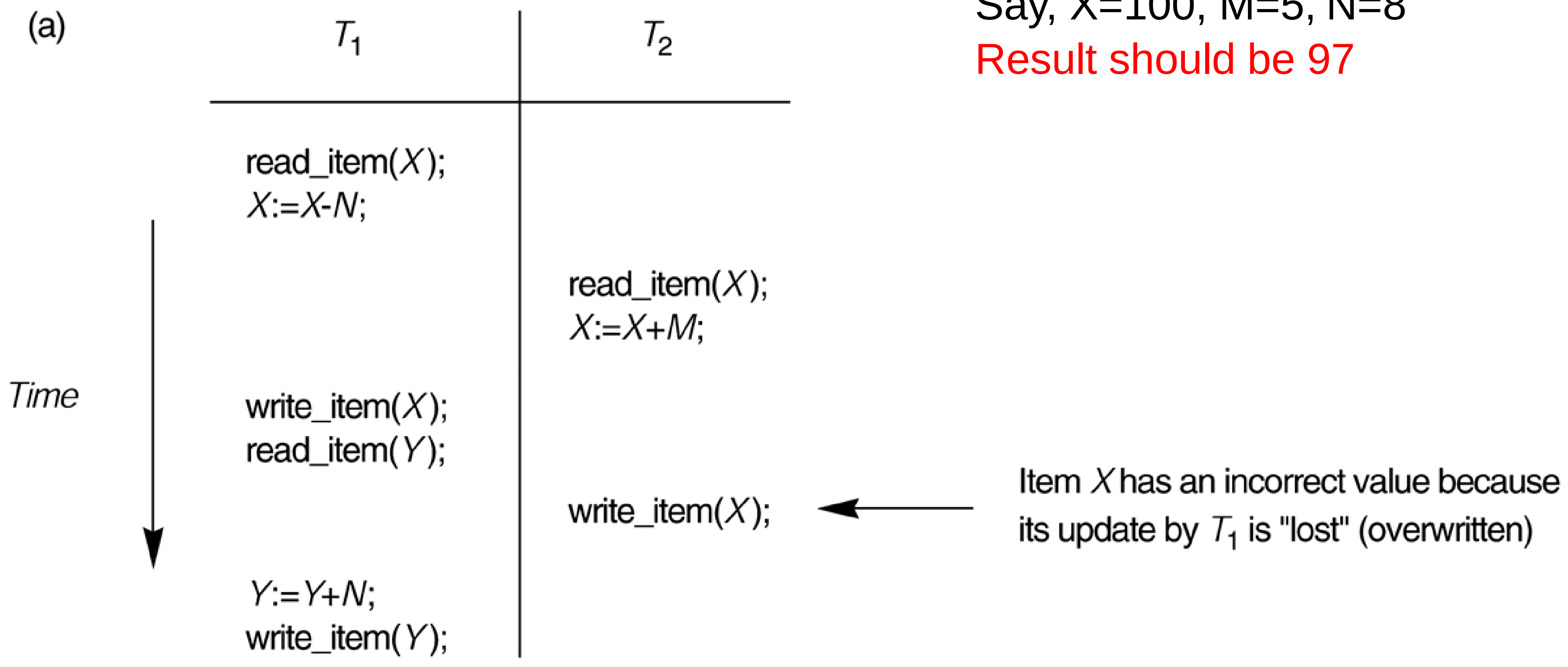
Why Concurrency Control is needed

- **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

Why Concurrency Control is needed

- **Lost Update Problem**



Say, X=100, M=5, N=8

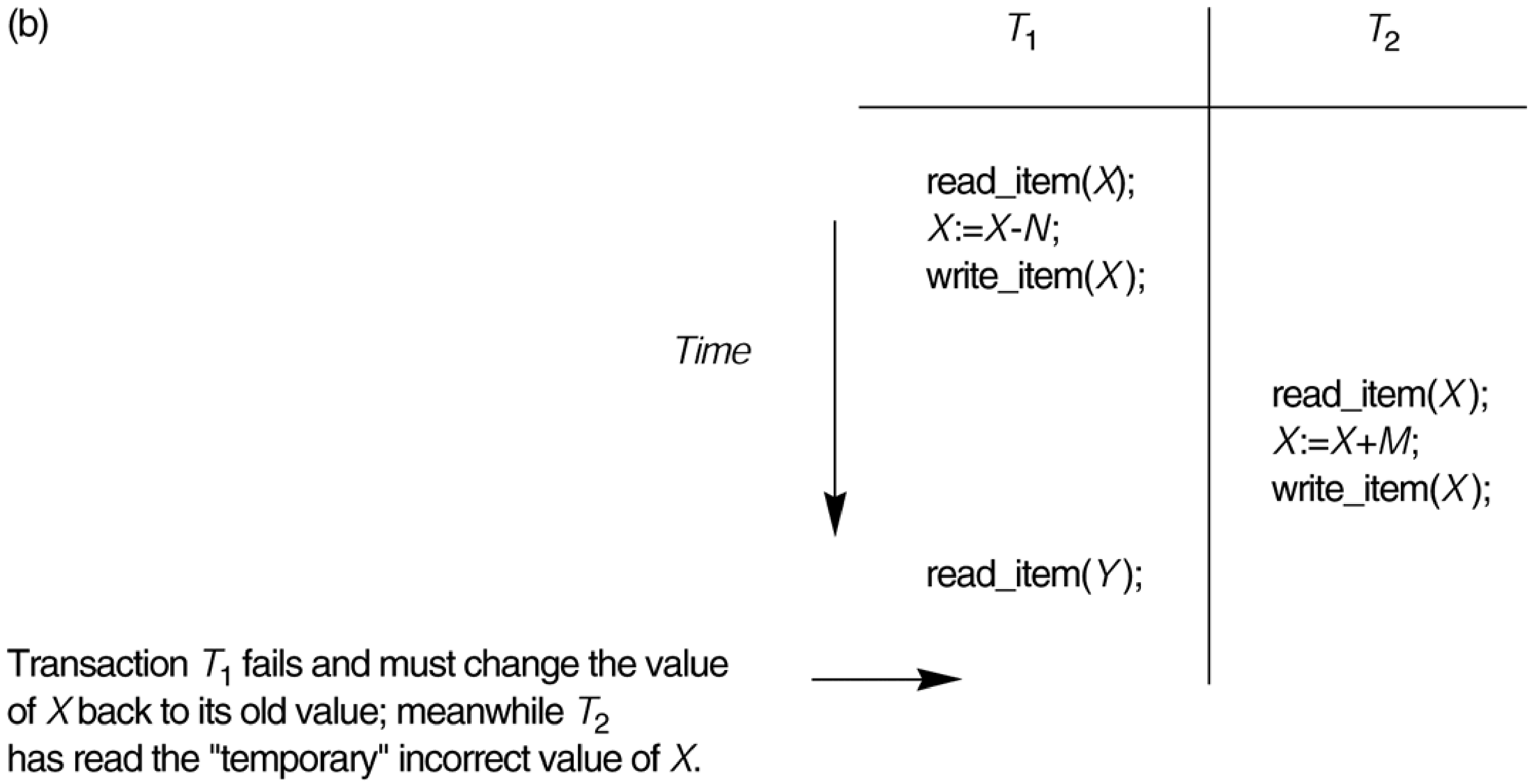
Result should be 97

Why Concurrency Control is needed

- **Temporary Update (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

Why Concurrency Control is needed

- **Temporary Update (Dirty Read) 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.

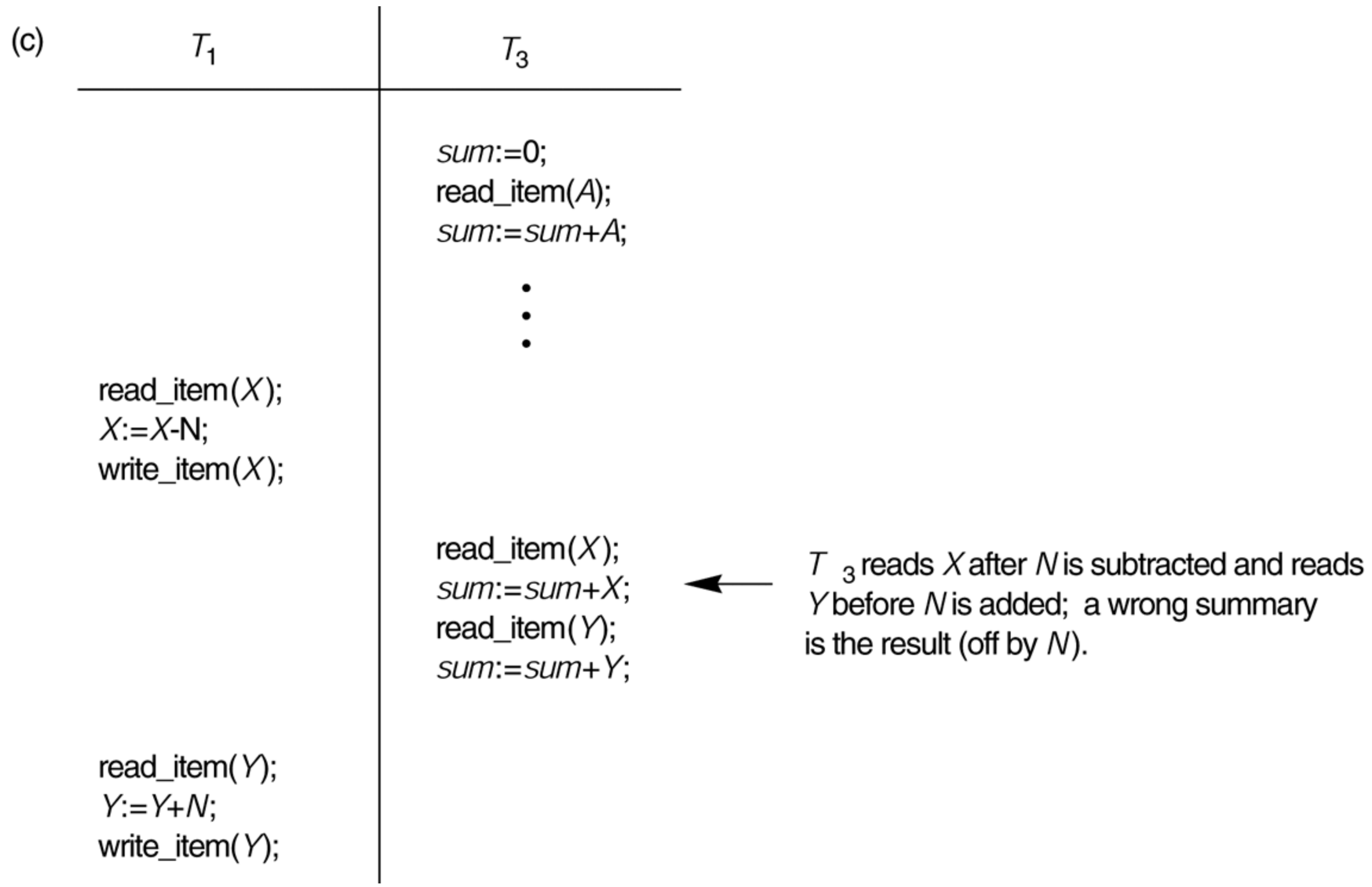
Why Concurrency Control is needed

- **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 Concurrency Control is needed

❖ 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*;
		← 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_item(*Y*);
Y:=*Y*+*N*;
write_item(*Y*);		

Why Concurrency Control is needed

- **Non-repeatable read**

  Transaction T reads an item twice and the item is changed by another transaction T’ between the two reads → T receives two different values for its two reads of the same item

  Eg. Airlines reservation system

Transaction Support in SQL

- **Dirty Read**
- **Non-repeatable read**
- **Phantom Read**

Significance of Recovery

- What causes a Transaction to fail
  - A Computer Failure (system crash)
  - A Transaction or System Error
  - Local Errors or Exception Conditions
  - Concurrency Control Enforcement
  - Disk Failure
  - Physical Problems and Catastrophes

ACID Properties

- **Atomicity**
- **Consistency**
- **Isolation**
- **Durability**

ACID Properties

A **transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

Atomicity: Either all operations of the transaction are properly reflected in the database or none are.

Consistency: Execution of a transaction in isolation preserves the consistency of the database.

Isolation: Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.

That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.

Durability: After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Example of Fund Transfer

- Transaction to transfer \$50 from account A to account B:

  1. **read**(A)
  2. $A := A - 50$
  3. **write**(A)
  4. **read**(B)
  5. $B := B + 50$
  6. **write**(B)

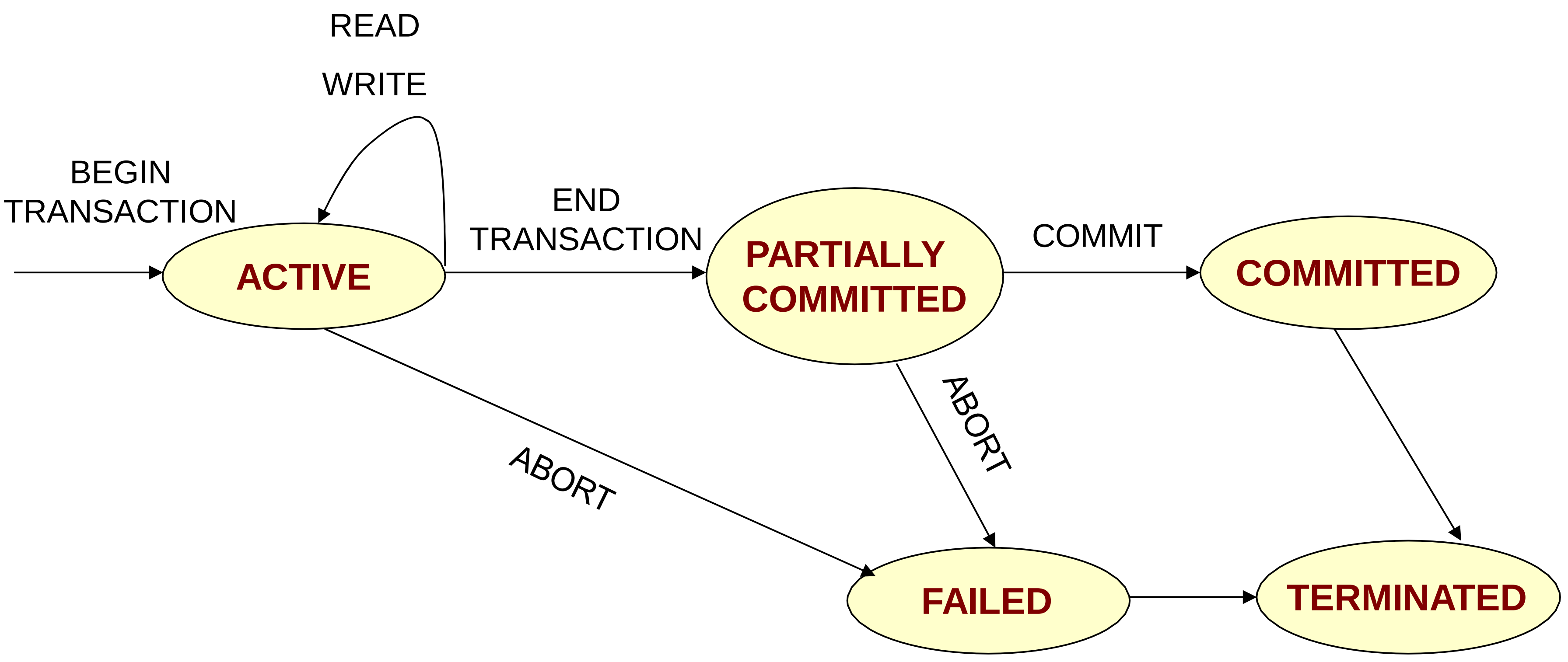
- if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result
- the sum of A and B should be unchanged by the execution of the transaction

Transaction States and Additional Operations

- A transaction is an atomic unit of work that is either completed in its entirety or not done at all
- Operations of a Transaction:
  - BEGIN TRANSACTION
  - READ OR WRITE (ACTIVE)
  - END TRANSACTION
  - COMMIT TRANSACTION / PARTIAL COMMIT
  - FAILED
  - ROLLBACK (ABORT)

State Transition Diagram

- States of Transaction Execution



System Log

- To be able to recover from failures that affect transaction
- System maintains a log to keep track of all transaction operation that affects the database item
- This is required for recovery from failure
- Log is kept on DISK → expected to be robust except catastrophic failure
- Log is periodically backed up to archival storage
- List of entries present in a log file → LOG RECORD

Log Records

- [start_transaction, T]
- [write_item, T, X, old_value, new_value]
- [read_item, T, X]
- [commit, T]
- [abort, T]

UNDO

REDO

Commit Point of a Transaction

- A transaction T reaches its commit point when all its operation that access the database item have been executed successfully
- Effect of all the transaction operations on the database have been recorded in the LOG
- Beyond the commit point transaction is said to be committed and is permanently recorded on database

How Log Records and Database are Affected on the Execution of any Query

- How Log records and data base are affected on the execution of the following query?
- Query1:

```
UPDATE saving SET balance=500 where accno=1;
COMMIT;
```

- Query2:

```
UPDATE saving SET balance=500 where accno=1;
ROLLBACK;
```

Steps of Transaction Execution

Step1: The system will check about the availability of the table as well

as the attributes

Step2: Whether there is any indexing/ hashing available on search key

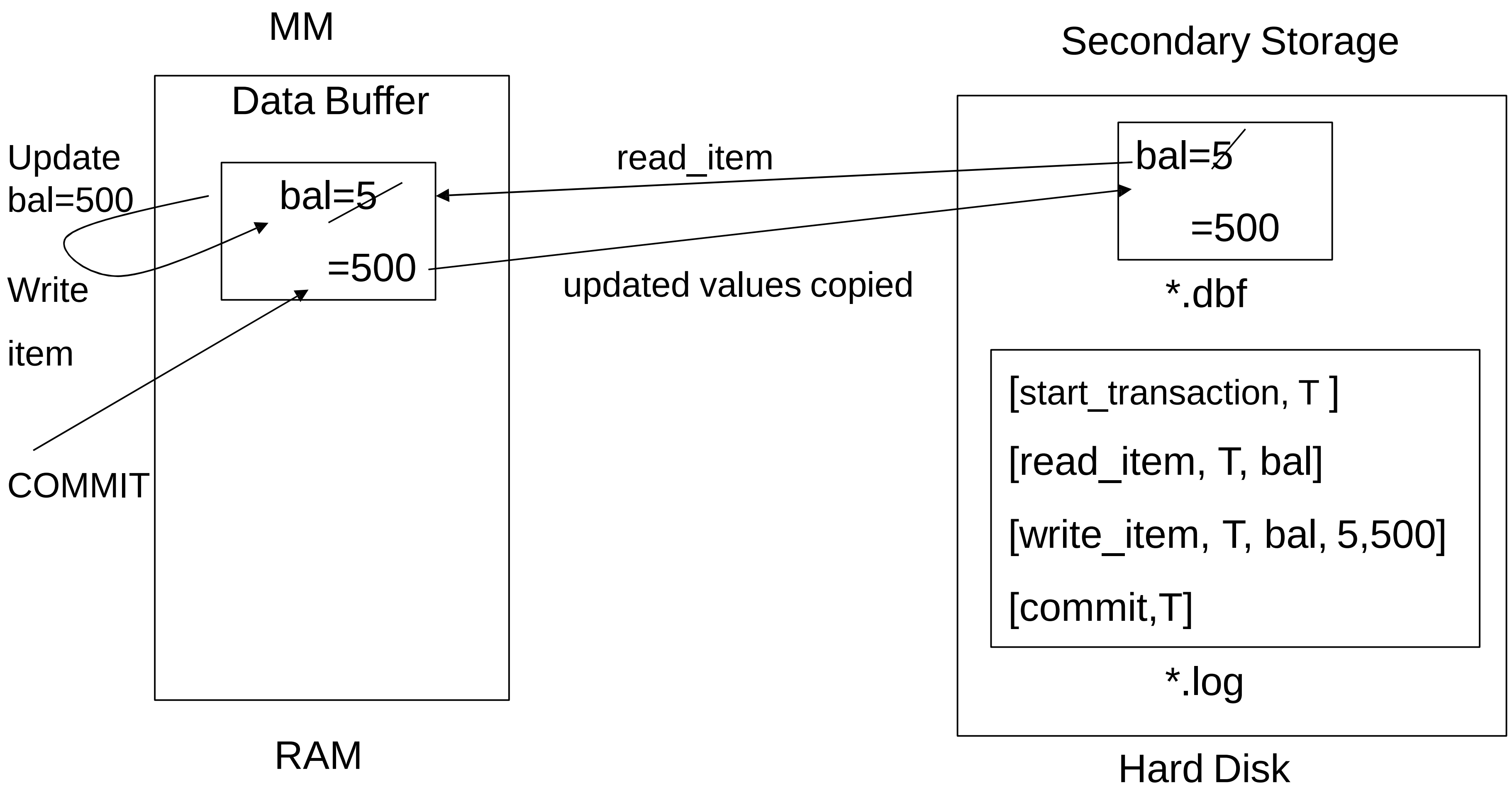
Step3: If index exists search and retrieve the data accordingly

Else retrieve the data based on search key values only

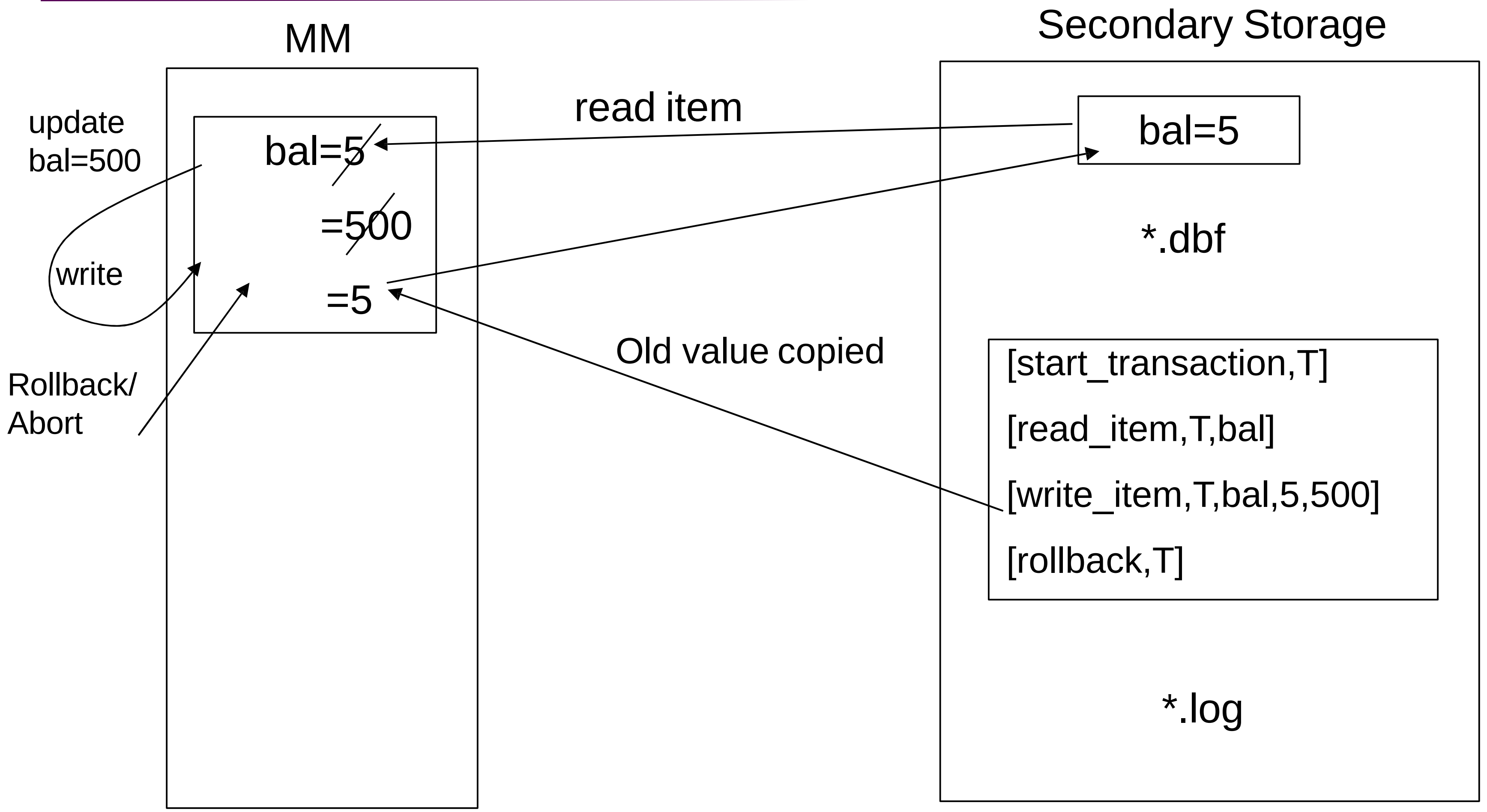
Step4: Update the database in MM and LOG as shown in the figure

Step4: COMMT/ROLLBACK as required

Steps of Transaction Execution: Query1



Steps of Transaction Execution: Query2



Schedule

- When transactions are executing concurrently in an interleaved fashion then the order of execution of operations from the various transactions is known as a schedule
- A schedule (history) S of n transactions $T_1, T_2, \ldots, T_n$ 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.
- Total order Schedule Vs. Partial Ordered Schedule

Example of Schedule

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

(a) Transaction T1

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

(b) Transaction T2

(a)

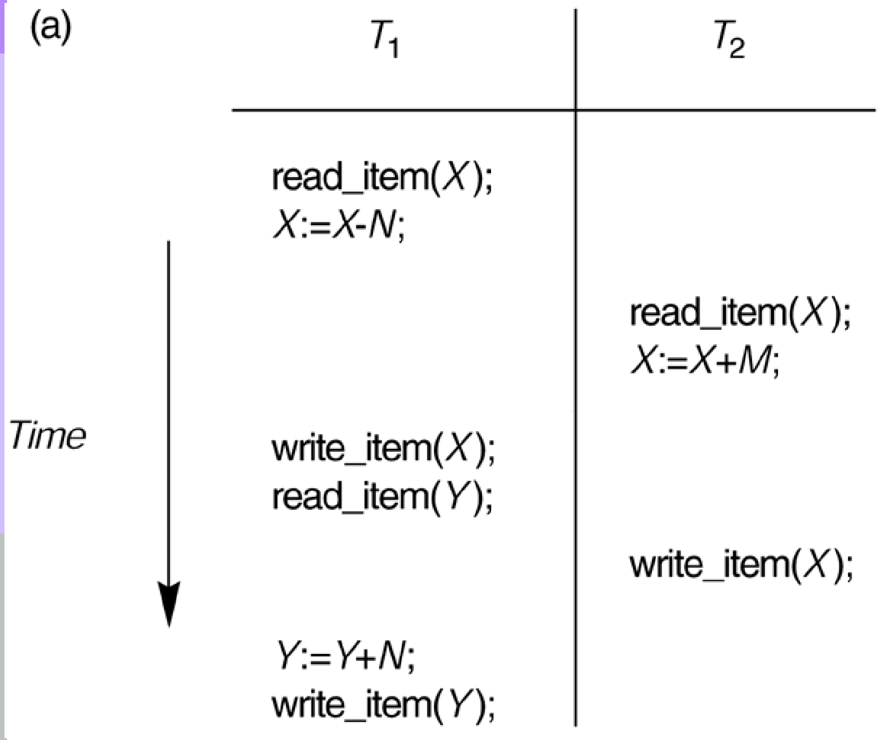
	T_1	T_2
Time ↓	read_item(X); X:=X-N;	
		read_item(X); X:=X+M;

	write_item(X);
	read_item(Y);	

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

Schedule S_a

Example of Schedule



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);	

read_item(x)→r(x)

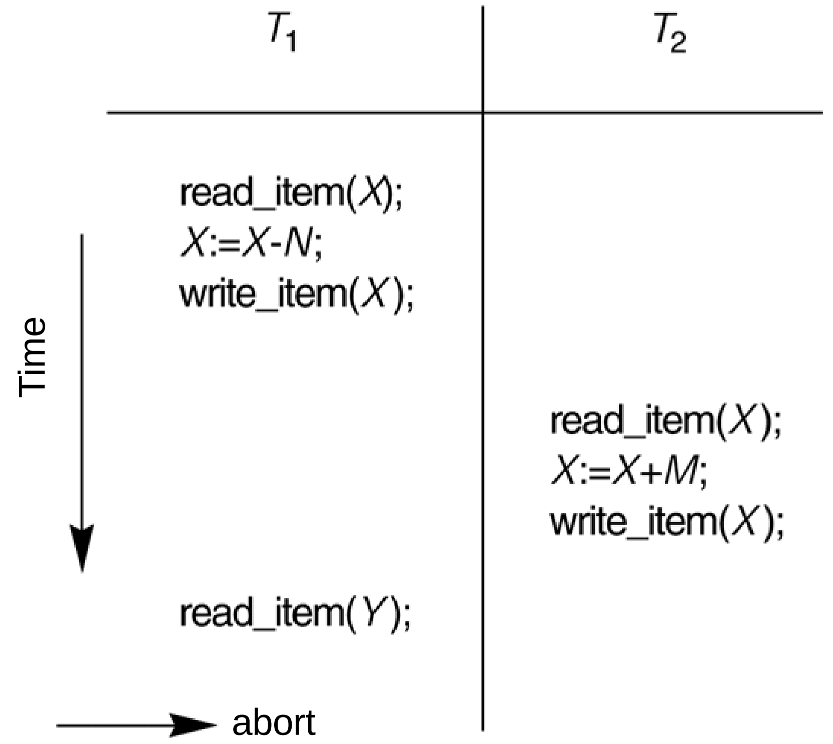
write_item(x)→ w(x)

Commit→ c;

Abort → a;

S_a: $r_1(x);r_2(x);w_1(x);r_1(y);w_2(x);w_1(y);$

Example of Schedule



S_b: $r_1(x)$; $w_1(x)$; $r_2(x)$; $w_2(x)$; $r_1(y)$;a_1;

Conflicting Operations in a Schedule

- Two operations in a schedule are said to conflict if they satisfy all the three of the following:
  - They belong to different transaction;
  - They access the same item x;
- At least one of them is a write_item(x);

For example, in schedule S_a

S_a: $r_1(x);r_2(x);w_1(x);r_1(y);w_2(x);w_1(y);$

1) $r_1(x);w_2(x);$
2) $r_2(x);w_1(x);$
3) $w_1(x);w_2(x);$

Conflicting Operations in a Schedule

- Test whether following operations conflict or not?

S_a: $r_1(x);r_2(x);w_1(x);r_1(y);w_2(x);w_1(y);$

For example, in schedule Sa:

1) $r_1(x); r_2(x);$ ……….
2) $r_1(x);w_1(x);$ ……….
3) $w_2(x); w_1(y);$………

Transaction Schedule

When transaction are executed concurrently in an interleaved fashion.

A schedule S of n transactions T_1, T_2,…, T_n:

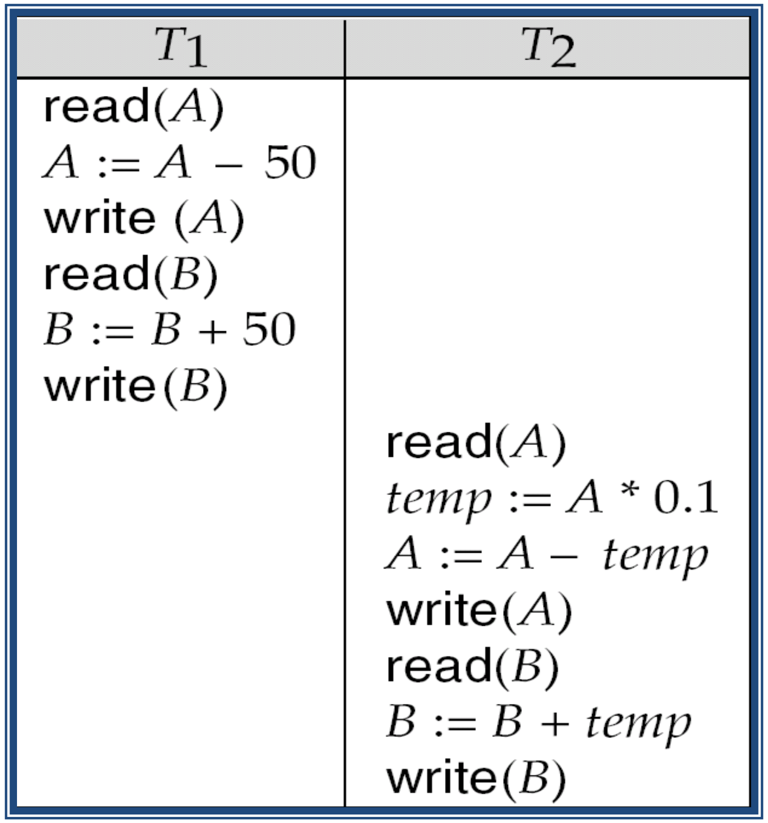
- All the instructions of the participating transactions must be present in S.
- The relative ordering of the instructions of each individual transaction must be preserved.
- However, the instructions of the different transactions may be in interleaving fashion.

A transaction that successfully completes its execution will have a commit instructions as the last statement (will be omitted if it is obvious).

A transaction that fails to successfully complete its execution will have an abort instructions as the last statement (will be omitted if it is obvious).

Schedule 1

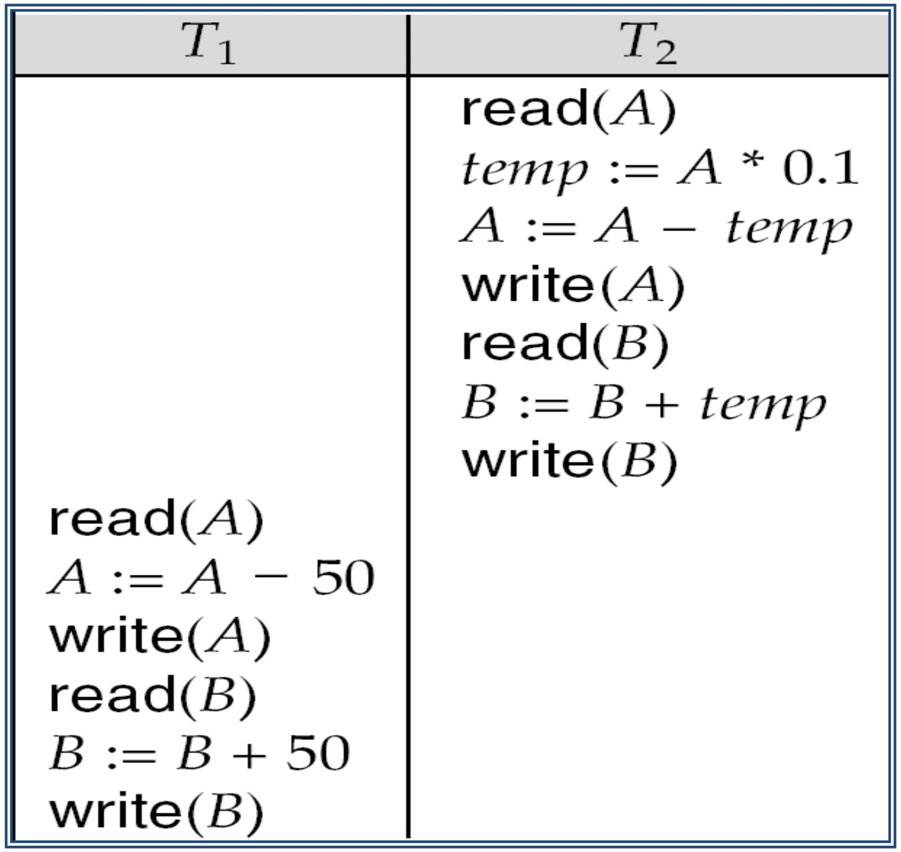
Let T_1 transfer $50 from *A* to *B*, and T_2 transfer 10% of the balance from *A* to *B.*
A serial schedule in which T_1 is followed by T_2:



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

Schedule 2

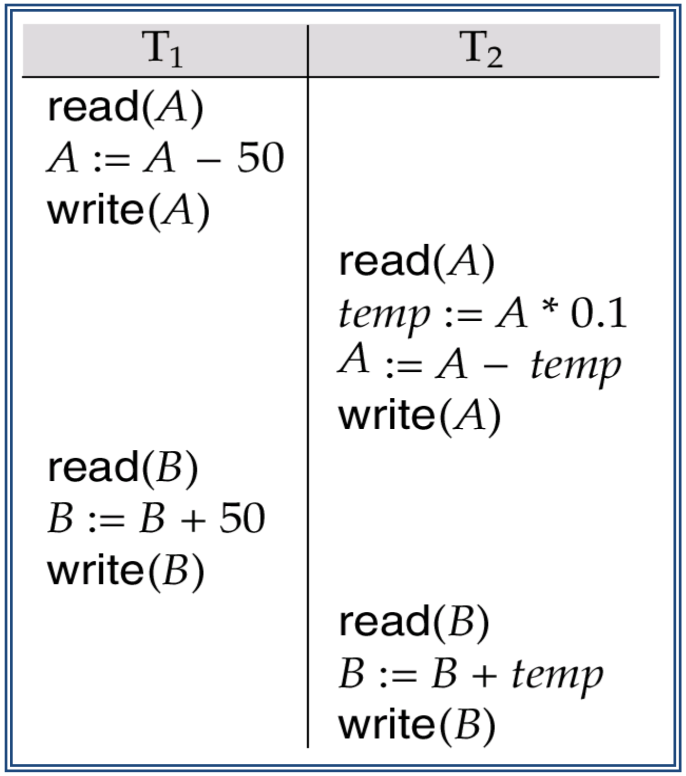
- A serial schedule where T_2 is followed by T_1



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

Schedule 3

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

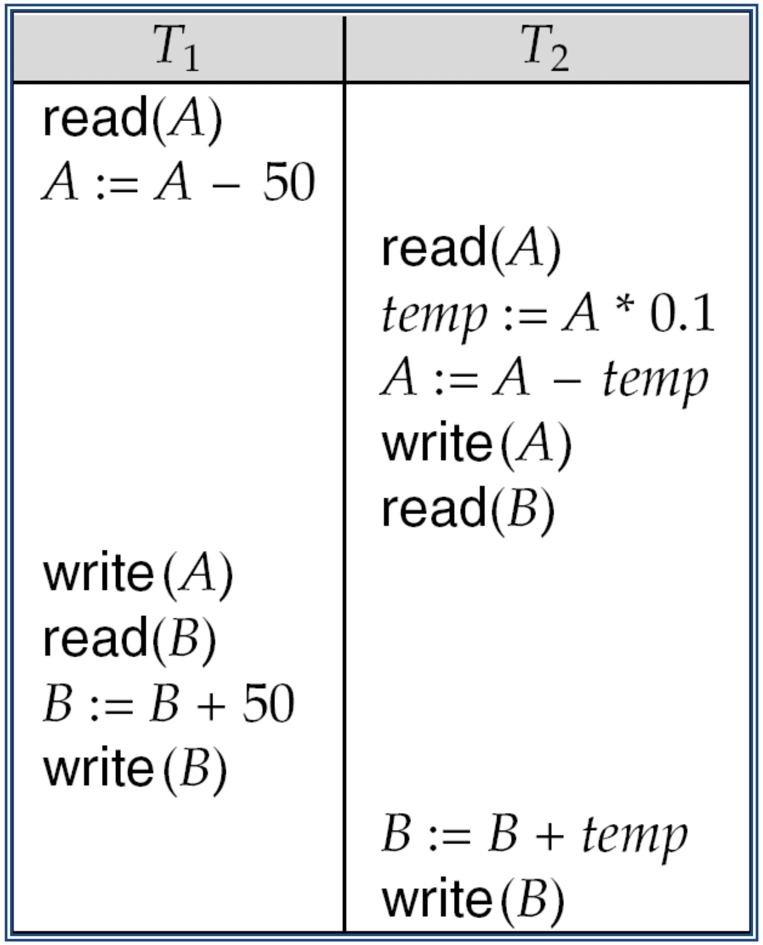


T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

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

Schedule 4

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



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

Serializability

- Basic Assumption: Each transaction preserves database consistency.
  - ✓ Thus serial execution of a set of transactions preserves database consistency.

- 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.

Serializability

- **Result equivalent:** Two schedules are called result equivalent if they produce the same final state of the database.
  - ✓ A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
  1. **conflict serializability**
  2. **view serializability**

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

Conflicting Instructions

- Instructions l_i and l_j of transactions T_i and T_j respectively, conflict if and only if there exists some item Q accessed by both l_i and l_j, and at least one of these instructions wrote Q.

1. l_i = read(Q), l_j = read(Q). l_i and l_j don't conflict.
2. l_i = read(Q), l_j = write(Q). They conflict.
3. l_i = write(Q), l_j = read(Q). They conflict
4. l_i = write(Q), l_j = write(Q). They conflict

- Intuitively, a conflict between l_i and l_j forces a (logical) temporal order between them.

  - If l_i and l_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflicting Serializable

- **Conflict equivalent:** Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both schedules.

- **Conflict serializable:** A schedule S is said to be conflict serializable if it is conflict equivalent to some serial schedule S'.

Conflict Serializability (Cont.)

Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions.

Therefore Schedule 3 is conflict serializable.

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

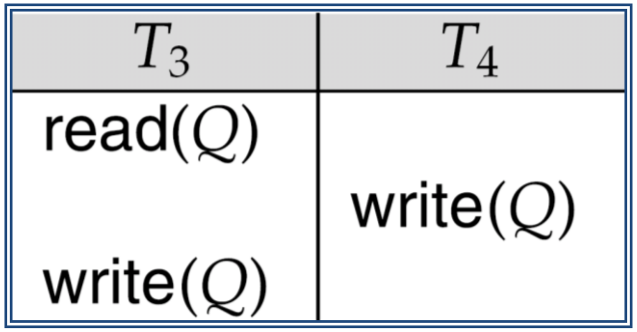
Schedule 3

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

Schedule 6

Conflict Serializability (Cont.)

Example of a schedule that is not conflict serializable:



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

We are unable to swap instructions in the above schedule to obtain either the serial schedule $< T_3, T_4 >$, or the serial schedule $< T_4, T_3 >$.

View Serializability

- Let *S* and *S´* be two schedules with the same set of transactions. *S* and *S´* are view equivalent if the following three conditions are met:

1. For each data item *Q,* if transaction T_i reads the initial value of *Q* in schedule *S,* then transaction T_i must, in schedule *S´*, also read the initial value of *Q.*

2. For each data item *Q* if transaction T_i executes **read**(*Q)* in schedule *S*, and that value was produced by transaction T_j (if any), then transaction T_i must in schedule *S´* also read the value of *Q* that was produced by transaction T_j .

3. For each data item *Q*, the transaction (if any) that performs the final **write**(*Q*) operation in schedule *S* must perform the final **write**(*Q*) operation in schedule *S´*.

As can be seen, view equivalence is also based purely on reads and writes alone.

View Serializability (Cont.)

- A schedule *S* is view serializable it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Conflict serializability is stricter than view serializability. With unconstrained write (or blind write), a schedule that is view serializable is not necessarily conflict serialiable.
- ✓ Below is a schedule which is view-serializable but *not* conflict serializable.

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

- ✓ What serial schedule is above equivalent to?
- ✓ Every view serializable schedule that is not conflict serializable has blind writes.

Testing for Serializability

Testing for conflict serializability

Algorithm:

1. Looks at only read_Item (X) and write_Item (X) operations
2. Constructs a precedence graph (serialization graph) - a graph with directed edges
3. An edge is created from T_i to T_j if one of the operations in T_i appears before a conflicting operation in T_j
4. The schedule is serializable if and only if the precedence graph has no cycles.

Example Schedule (Schedule A) + Precedence Graph

T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				

$$\begin{tikzcd}
T_1 \arrow[r, bend left] \arrow[d, bend right] \arrow[dr] & T_2 \arrow[d, bend left] \\
T_3 \arrow[r, bend right] & T_4
\end{tikzcd}$$

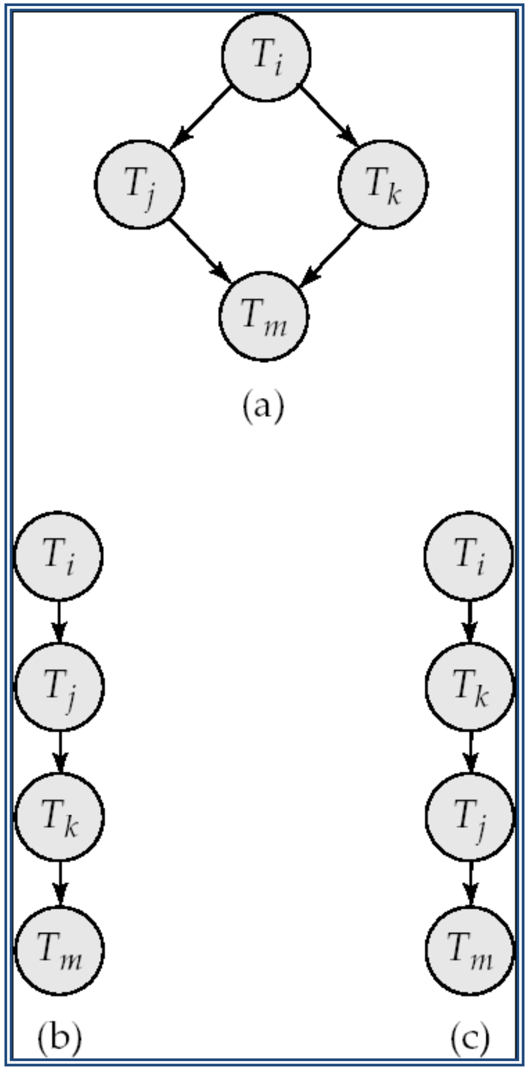
Test for Conflict Serializability

- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph.

  (Better algorithms take order $n + e$ where e is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.
  - ✓ This is a linear order consistent with the partial order of the graph.
  - ✓ For example, a serializability order for Schedule A would be

    $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$

    Are there others?



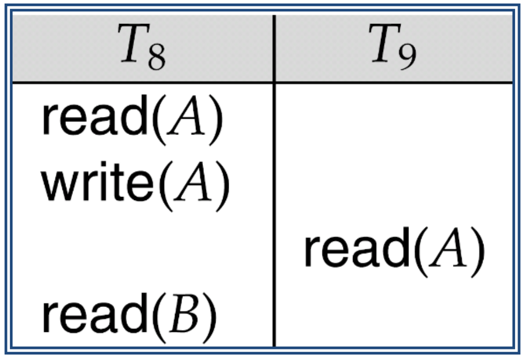
Test for View Serializability

- The precedence graph test for conflict serializability cannot be used directly to test for view serializability.
  - ✓ Extension to test for view serializability has cost exponential in the size of the precedence graph.
- The problem of checking if a schedule is view serializable falls in the class of *NP*-complete problems.
-
  - ✓ Thus existence of an efficient algorithm is *extremely* unlikely.
- However practical algorithms that just check some sufficient conditions for view serializability can still be used.

Recoverable Schedules

Need to address the effect of transaction failures on concurrently running transactions.

- ➢ Recoverable schedule — if a transaction T_j reads a data item previously written by a transaction T_i, then the commit operation of T_i appears before the commit operation of T_j.
- ✓ The following schedule (Schedule 11) is not recoverable if T_9 commits immediately after the read



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

- ✓ If T_8 should abort, T_9 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.

Cascading Rollbacks

Cascading rollback – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

If T_{10} fails, T_{11} and T_{12} must also be rolled back.
Can lead to the undoing of a significant amount of work

Cascadeless Schedules

Cascadeless schedules — cascading rollbacks cannot occur; for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.

Every cascadeless schedule is also recoverable

It is desirable to restrict the schedules to those that are cascadeless

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - ✓ either conflict or view serializable, and
  - ✓ are recoverable and preferably cascadeless

- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
  - ✓ Are serial schedules recoverable/cascadeless?

- Testing a schedule for serializability *after* it has executed is a little too late!
- Goal – to develop concurrency control protocols that will assure serializability.

Concurrency Control vs. Serializability Tests

- Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless.
- Concurrency control protocols generally do not examine the precedence graph as it is being created
  - ✓ Instead a protocol imposes a discipline that avoids nonseralizable schedules.
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.
- Tests for serializability help us understand why a concurrency control protocol is correct.

Complete Schedule

- A schedule S of n transactions $T_1,T_2,\ldots,T_n$ is said to be a complete schedule if the following conditions hold:
  - The operations in S are exactly those operations in $T_1,T_2,\ldots,T_{n,}$ including a commit or abort operation as the last operation for each transaction in the schedule
  - 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 the schedule

Thank You

?