Transactions

Transactions

- Transaction Concept
- Transaction State
- Concurrent Executions
- Serializability
- Implementation of Isolation

Introduction to 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.

Introduction to Transaction Processing (Contd..)

- 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

Introduction to Transaction Processing (Contd..)

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

- **A database** is a collection of named data items
- **Granularity** of data - a field, a record , or a whole disk block
- 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.

Introduction to Transaction Processing

READ AND WRITE OPERATIONS:

- 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_item(X) command includes the following steps:
  - Find the address of the disk block that contains item X.
  - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
  - Copy item X from the buffer to the program variable named X.

Introduction to Transaction Processing

READ AND WRITE OPERATIONS (contd.):

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

Two sample transactions

- FIGURE 17.2 Two sample transactions:
  - (a) Transaction T1
  - (b) Transaction T2

(a) T_1

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

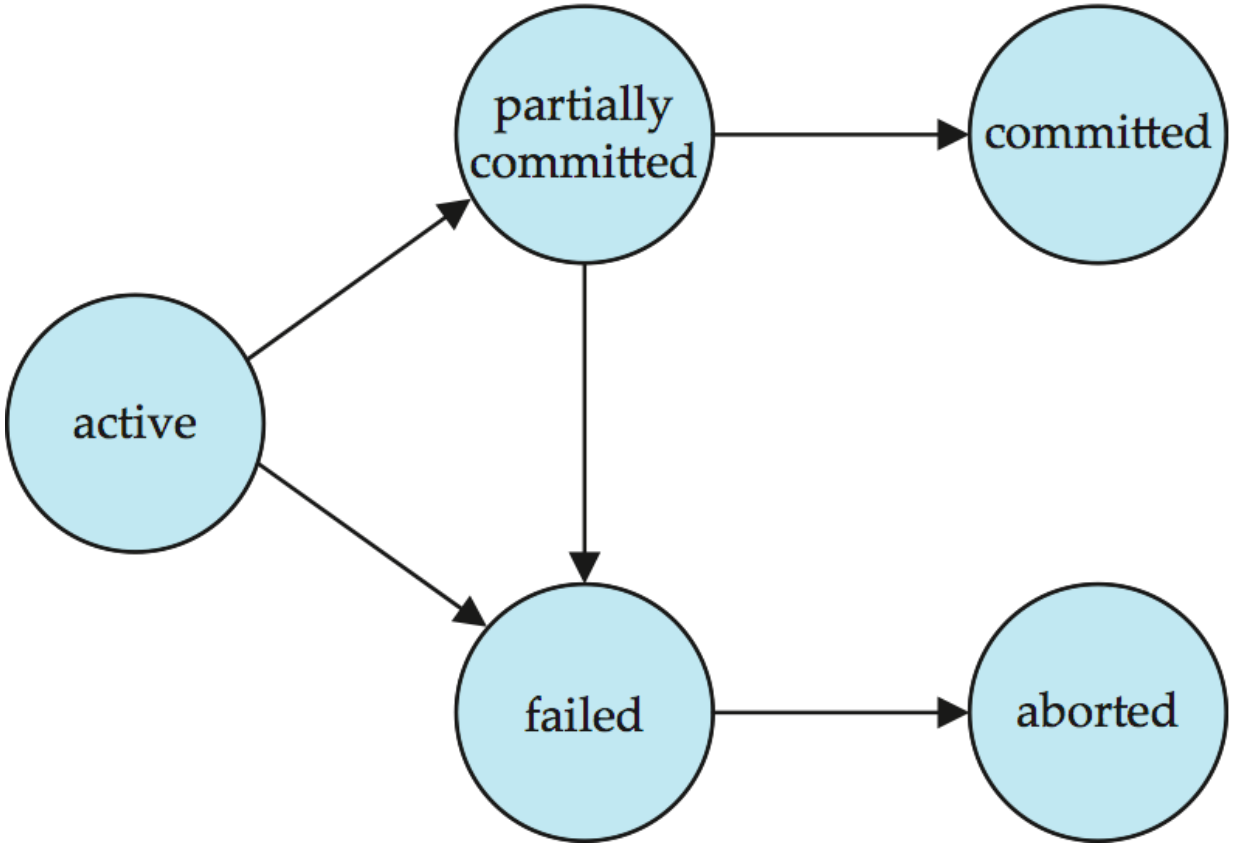
(b) T_2

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

Transaction State

- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed** -- after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - restart the transaction
    - can be done only if no internal logical error
  - kill the transaction
- **Committed** – after successful completion.

Transaction State (Cont.)



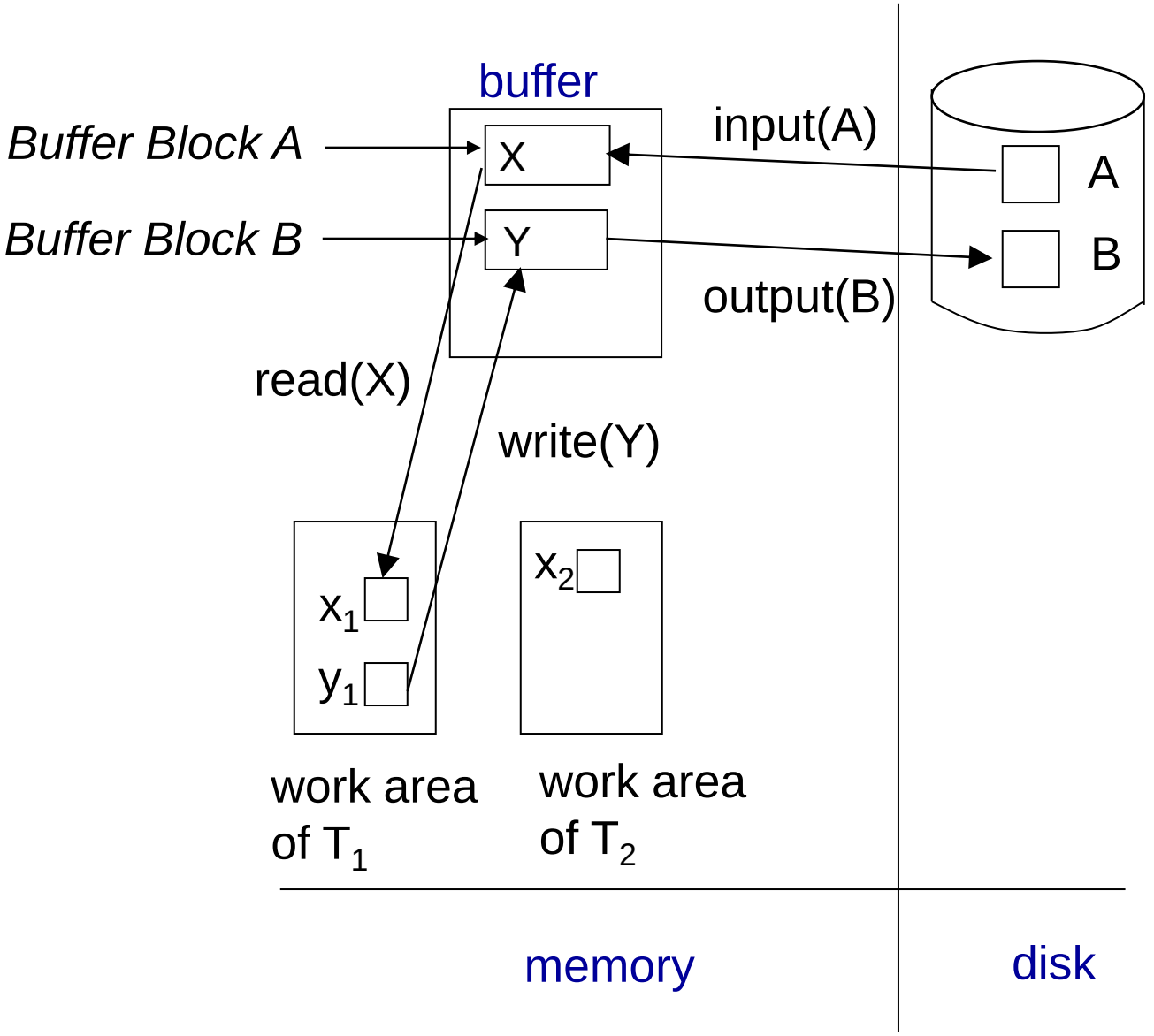
Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- E.g. 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*)
- Two main issues to deal with:
  - **Failures** of various kinds, such as hardware failures and system crashes
  - **Concurrent** execution of multiple transactions

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*)
- **Atomicity requirement**
  - **if the transaction fails** after step 3 and before step 6, **money will be "lost"** leading to an inconsistent database state
    - Failure could be due to software or hardware
  - the **system should ensure** that updates of a partially executed **transaction** are **not reflected in the database**
- **Durability requirement** — 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** even if there are software or hardware failures.

Example of Data Access



Example of Fund Transfer (Cont.)

- 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*)
- **Consistency requirement** in above example:
  - the **sum of A and B is unchanged** by the execution of the transaction
- In general, consistency requirements include
  - **Explicitly** specified **integrity constraints** such as primary keys and foreign keys
  - **Implicit** integrity constraints
    - e.g. **sum of balances of all accounts**, minus sum of loan amounts **must equal value of cash-in-hand**
  - A transaction must see a consistent database.
  - During transaction execution the database may be temporarily inconsistent.
  - When the transaction completes successfully the database must be consistent
    - Erroneous transaction logic can lead to inconsistency

Example of Fund Transfer (Cont.)

- **Isolation requirement** — if between steps 3 and 6, another transaction T2 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).

T1	T2
1. **read**(A)	
2. $A := A - 50$	
3. **write**(A)	
	read(A), read(B), print(A+B)
4. **read**(B)	
5. $B := B + 50$	
6. **write**(B	

- Isolation can be ensured trivially by running transactions **serially**
  - that is, one after the other.
- **However**, executing multiple transactions concurrently has **significant benefits**, as we will see later.

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.

Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **increased processor and disk utilization**, leading to better transaction *throughput*
    - E.g. one transaction can be using the CPU while another is reading from or writing to the disk
  - **reduced average response time** for transactions: short transactions need not wait behind long ones.
- **Concurrency control schemes** – mechanisms to achieve isolation
  - that is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database
    - Will study in Chapter 16, after studying notion of correctness of concurrent executions.

Schedules

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a commit instructions as the last statement
  - by default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

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)	
commit	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit

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) commit
read (A) $A := A - 50$ write (A) read (B) $B := B + 50$ write (B) commit	

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)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

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)	
commit	
	$B := B + temp$
	write (B)
	commit

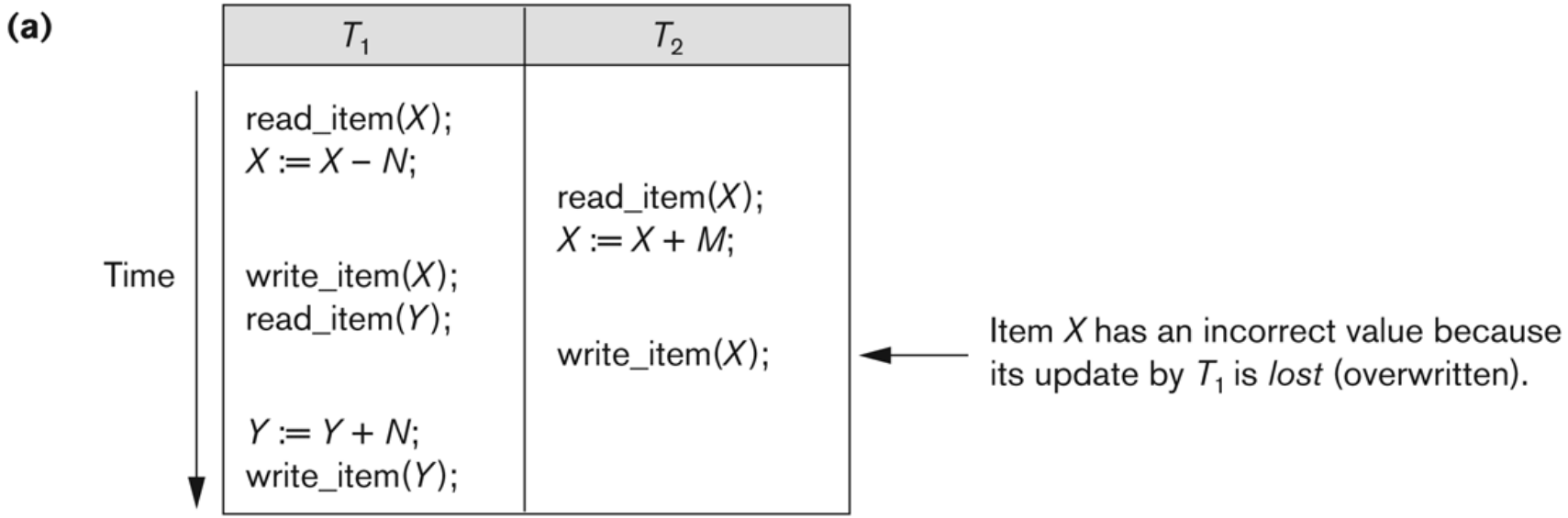
Transaction Processing

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 (see Section 17.1.4).
  - The updated item is accessed by another transaction before it is changed back to its original value.
- **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.

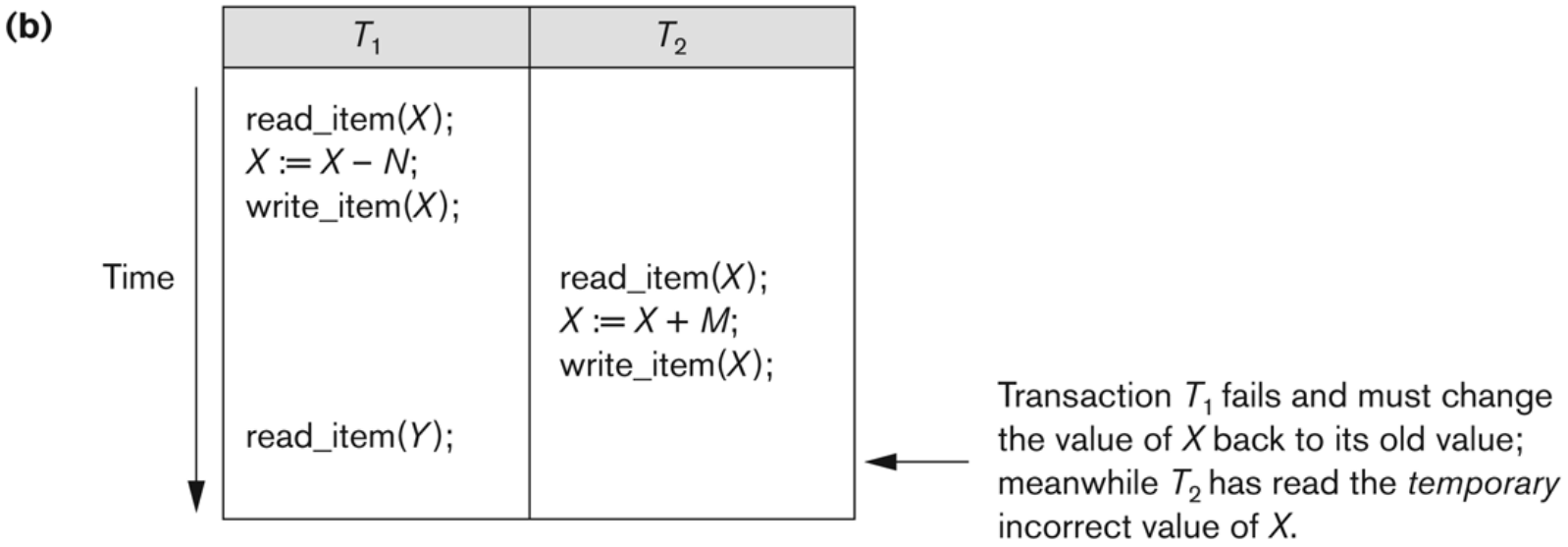
Concurrent execution is uncontrolled: (a) The lost update problem.

Figure 17.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.



Concurrent execution is uncontrolled: (b) The temporary update problem.

Figure 17.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.



Concurrent execution is uncontrolled: (c) The incorrect summary problem.

Figure 17.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) 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);
	$sum := sum + Y$;

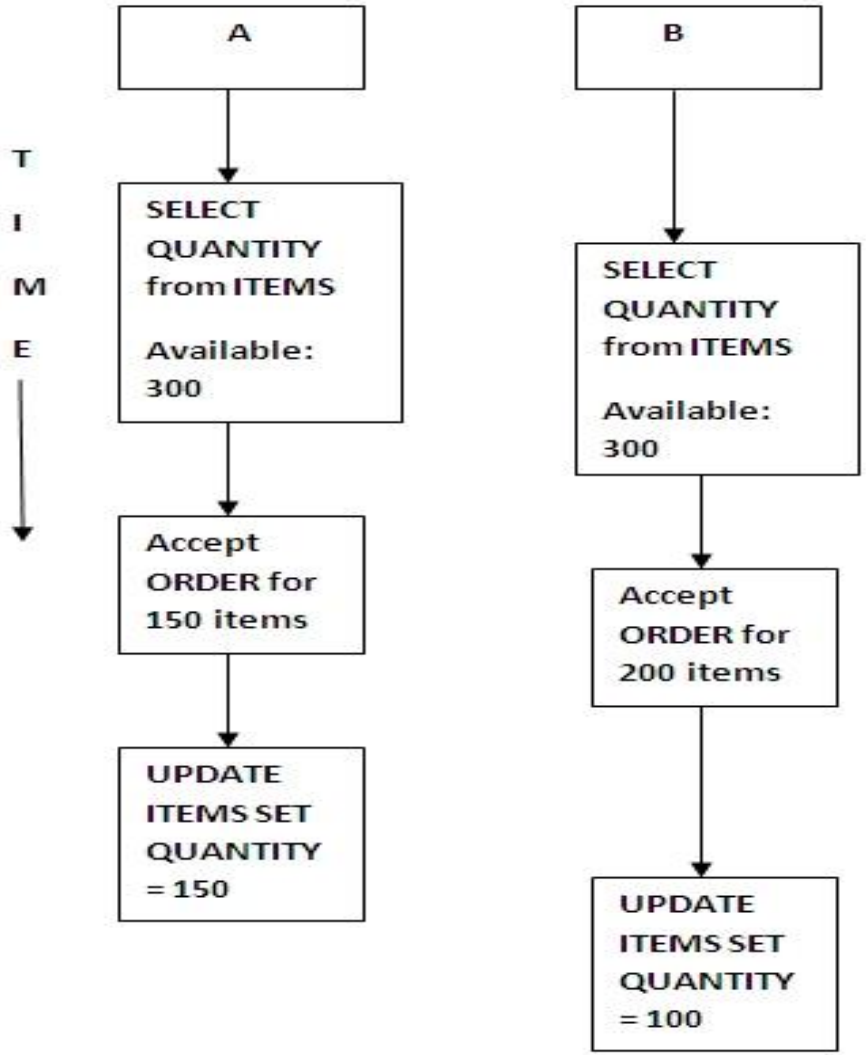
read_item(Y);
$Y := Y + N$;
write_item(Y);	

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

Example for The Lost Update Problem

- **Lost update problem:** A lost update is a typical problem in transaction processing in SQL. It happens when two queries access and update the same data from a database. This problem can be understood by the below given diagram. Here, A is processing an order for a client for 150 items. He checks from the ITEMS table that there are 300 available items. So he starts placing the order. After few seconds B gets an order for 200 items. He also checks the items table and finds that there are 300 items available. So he also starts placing an order. Meanwhile A confirms the order for 150 items and updates the ITEMS table and sets the quantity to 150. A few seconds later B confirms the order and updates the ITEMS table and sets the quantity to 100. This problem is known as lost update problem because both the orders of the users A and B have been accepted, but there is not enough items available. Hence, the updates are lost.

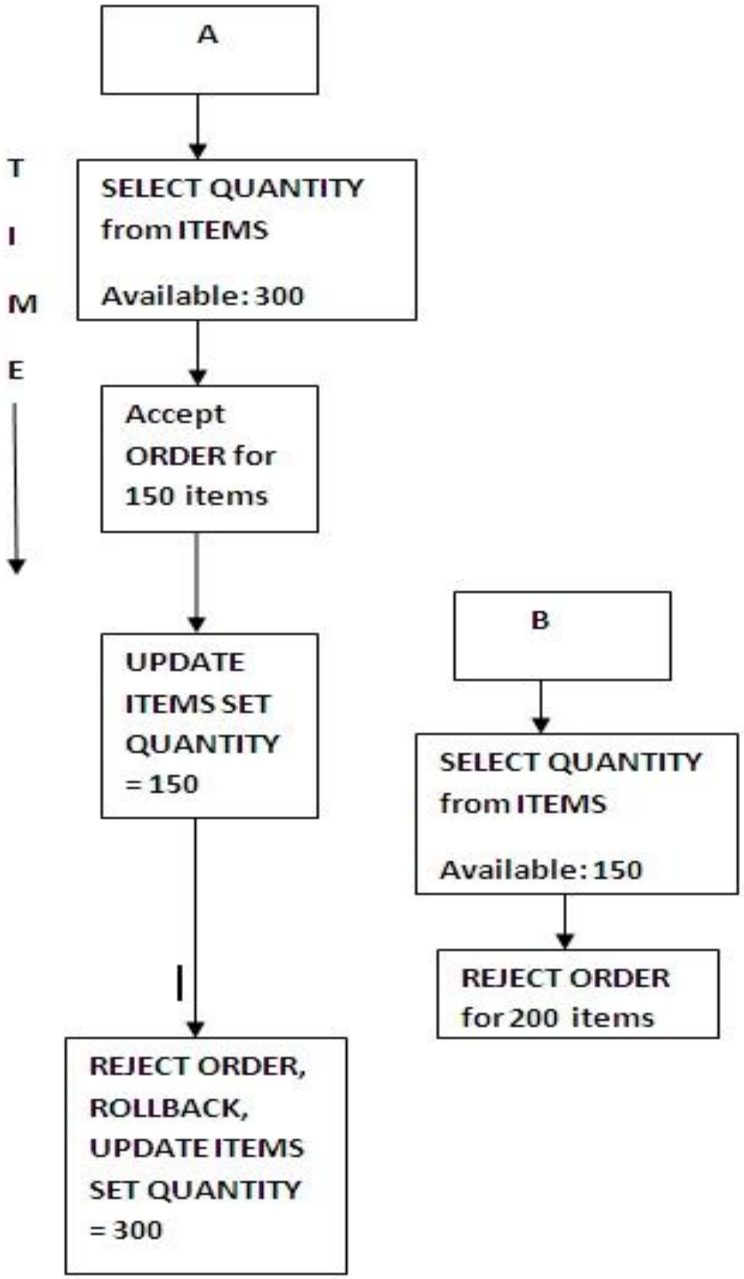
Example for The Lost Update Problem



Example for Uncommitted or Dirty read Problem

- In a dirty read problem, A is processing an order for a client for 150 items. He checks from the ITEMS table that there are 300 available items. So he starts placing the order. A confirms the order for 150 items and updates the ITEMS table and sets the quantity to 150. Now, B receives an order for 200 items. He checks the ITEMS table to find that there is not enough inventories (150 available) and rejects the order. By using business rules and transactions a note is sent informing that more items are required. Now, due to some reason client asks A to cancel the order, so A cancels the order, rolls back and updates the ITEMS table back to 300 items. This problem is known as dirty read because B saw the uncommitted update of A.

Example for Uncommitted or Dirty Read



Serializability

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

Simplified view of transactions

- We **ignore** operations **other** than **read** and **write** instructions
- 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 I_i and I_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both I_i and I_j, and at least one of these instructions wrote Q.

  1. I_i = **read**(Q), I_j = **read**(Q). I_i and I_j **don't conflict**.
  2. I_i = **read**(Q), I_j = **write**(Q). They **conflict**.
  3. I_i = **write**(Q), I_j = **read**(Q). They **conflict**
  4. I_i = **write**(Q), I_j = **write**(Q). They **conflict**

- Intuitively, a conflict between I_i and I_j forces a (logical) temporal order between them.
  - If I_i and I_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.

Conflict Serializability

- If a schedule *S* can be transformed into a schedule *S´* by a series of swaps of non-conflicting instructions, we say that *S* and *S´* are **conflict equivalent**.
- We say that a schedule *S* is **conflict serializable** if it is conflict **equivalent to a serial schedule**

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

Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts, "**dirty reads**"):

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

Anomalies with Interleaved Execution

- Unrepeatable Reads (RW Conflicts):

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

Anomalies (Continued)

- Overwriting Uncommitted Data (WW Conflicts):

T_1	T_2
write (A)	
	write (A)
	write (B)
	C
write (B)	
C	

Thank You