Transactions

UNIT IV

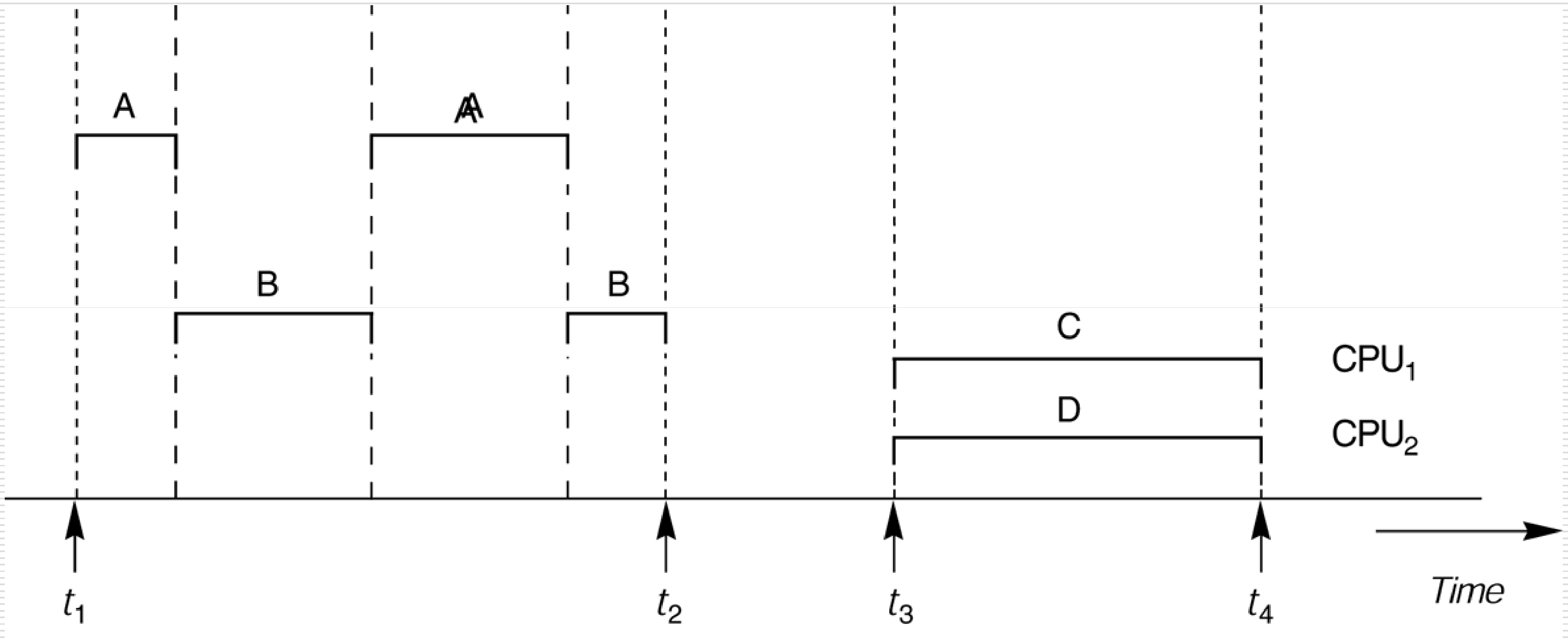
Overview

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

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.

Interleaved processing vs. parallel processing of concurrent transactions



Introduction to 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 is a unit of program execution that accesses and possibly updates various data items.
- 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.

Example

- Ex: 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

Introduction to Transaction Processing

SIMPLE MODEL OF A DATABASE:

- A database is a 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.

Introduction to Transaction Processing

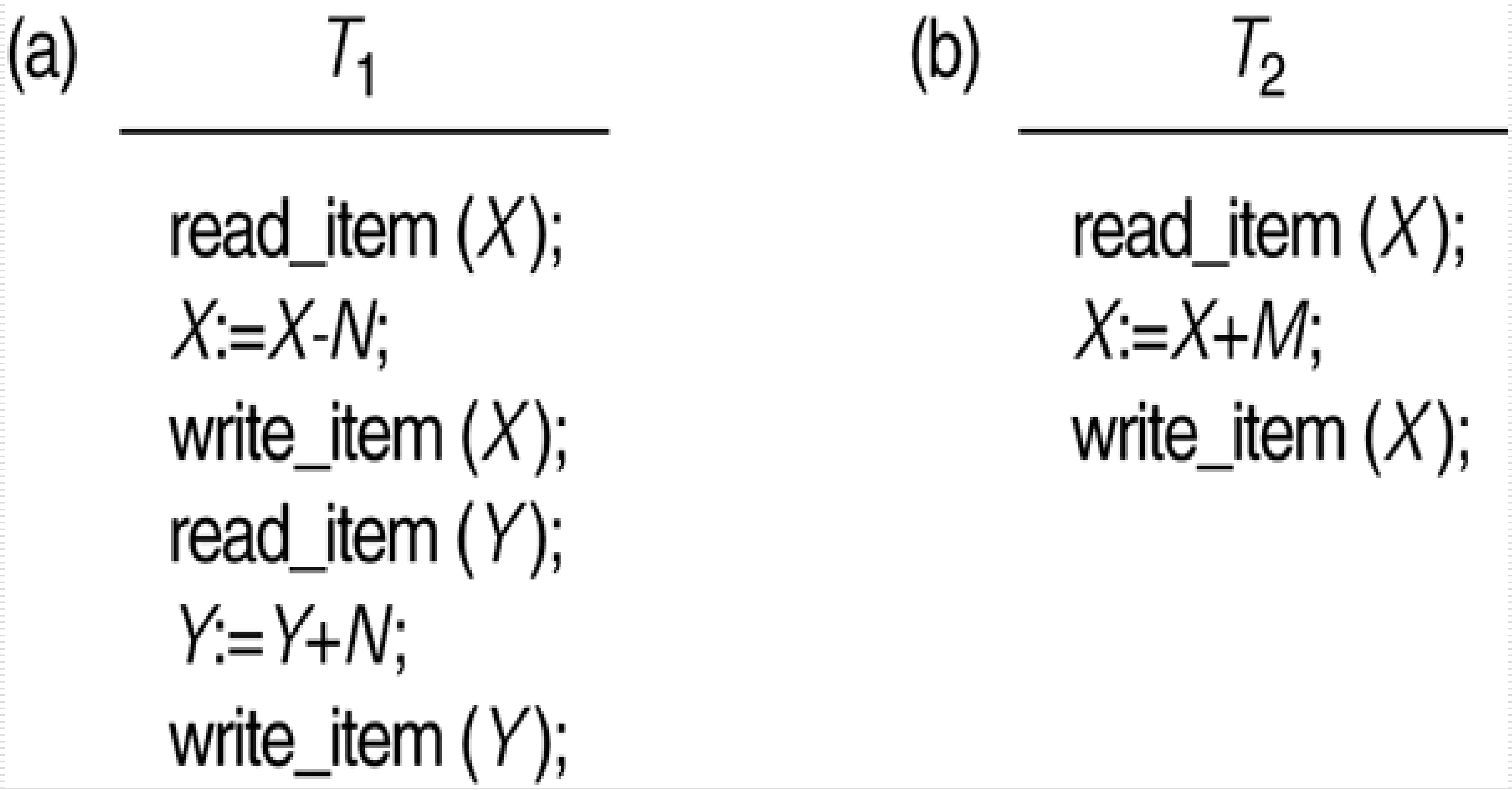
READ AND WRITE OPERATIONS:

- 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 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

- **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



(a) T_1

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

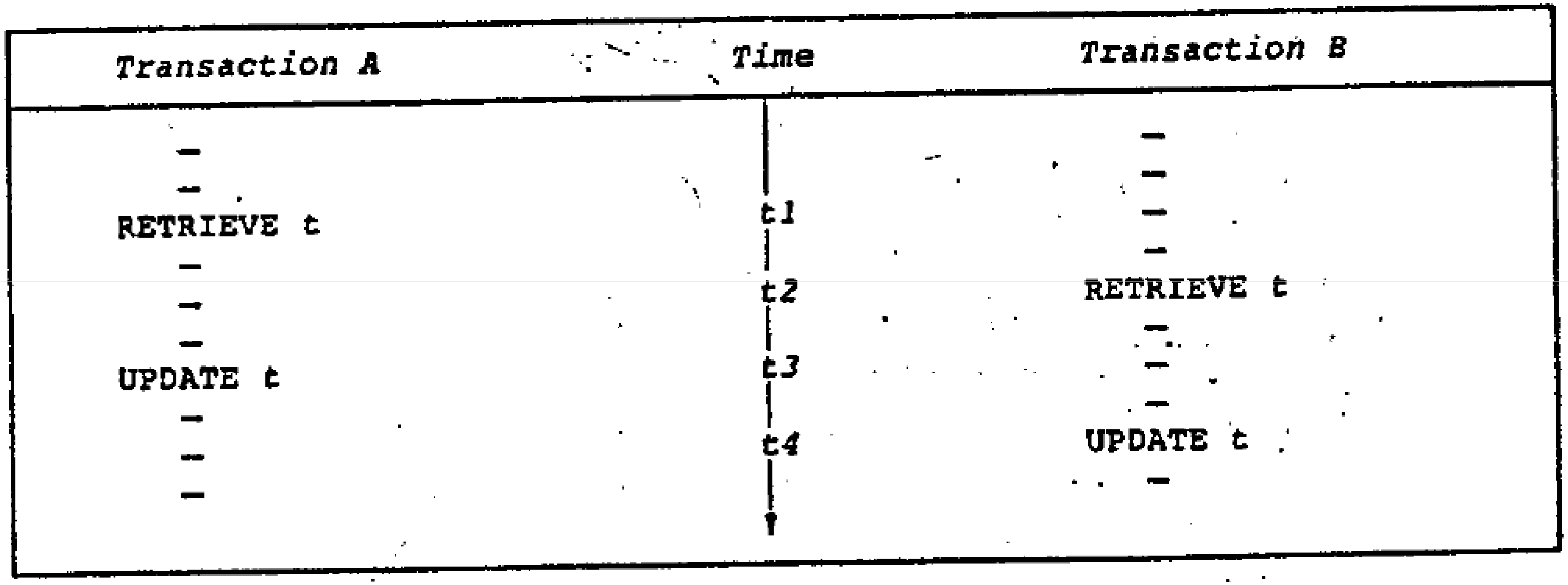
(b) T_2

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

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.

Lost update problem

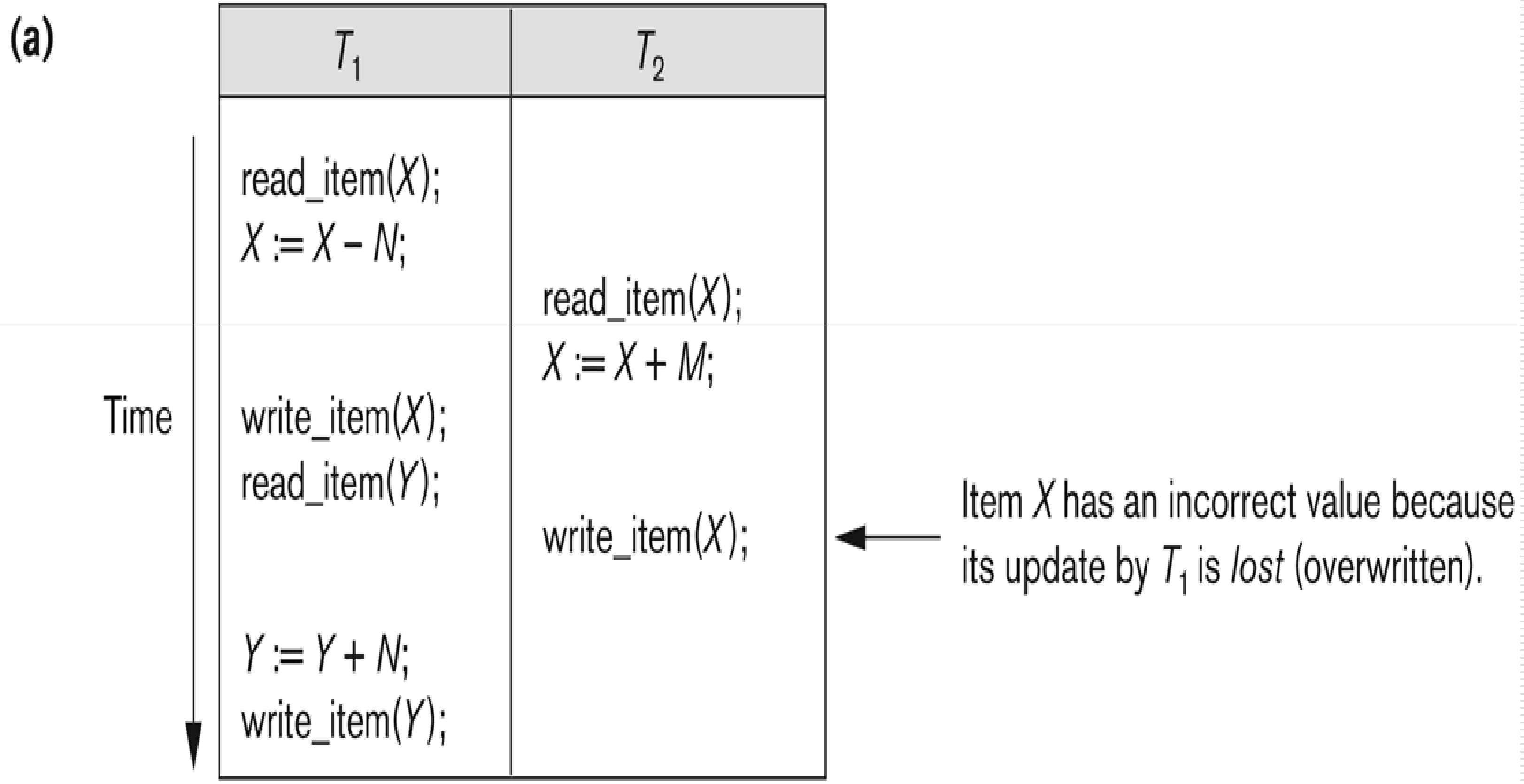


Transaction A	Time	Transaction B
—		—
—		—
RETRIEVE t	t1	—
—		—
—	t2	RETRIEVE t
—		—
UPDATE t	t3	—
—		—
—	t4	UPDATE t
—		—

Transaction *A* loses an update at time *t4* .

The lost update problem.

(a)



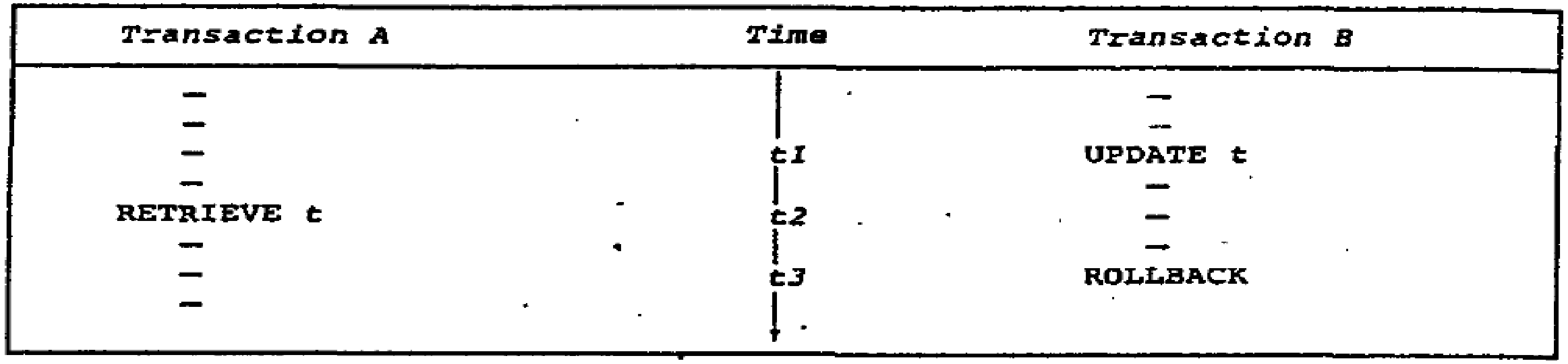
T_1	T_2
read_item(X);	
$X := X - N$;	
	read_item(X);
	$X := X + M$;
write_item(X);	
read_item(Y);	
	write_item(X);
$Y := Y + N$;	
write_item(Y);	

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

Why Concurrency Control is needed

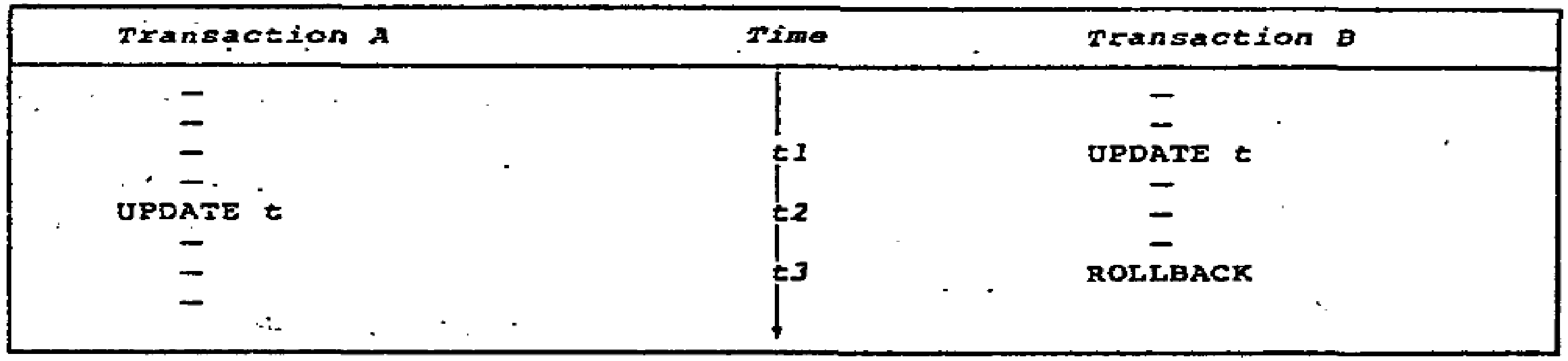
- The Temporary Update (or Dirty Read) Problem
  - This occurs when one transaction updates a database item and then the transaction fails for some reason.
  - The updated item is accessed by another transaction before it is changed back to its original value.

Uncommitted dependency problem



Transaction A	Time	Transaction B
—		—
—		—
—	t1	UPDATE t
—		—
RETRIEVE t	t2	—
—		—
—	t3	ROLLBACK
—		

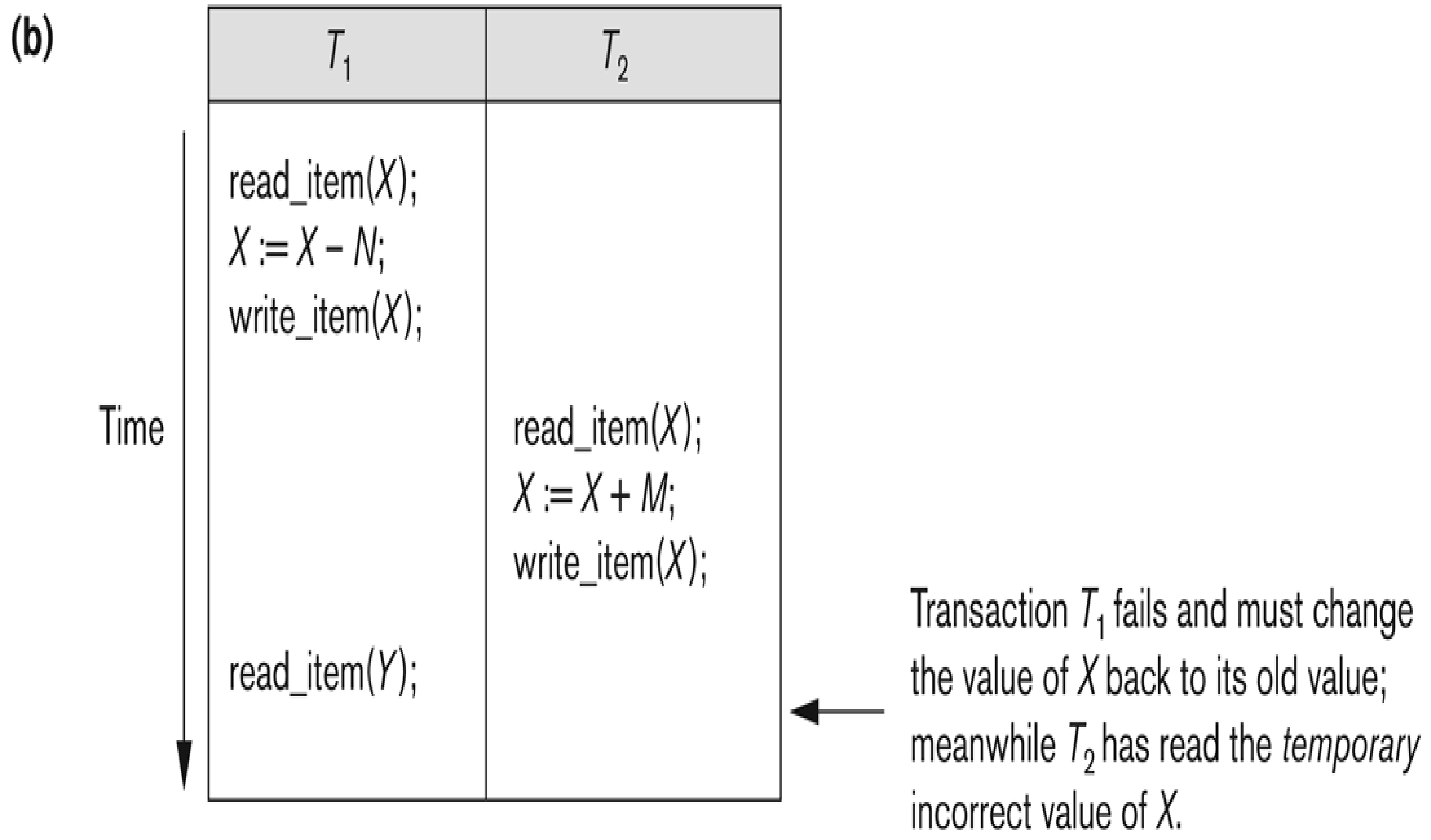
Fig. 16.2 Transaction *A* becomes dependent on an uncommitted change at time *t2*



Transaction A	Time	Transaction B
—		—
—		—
—	t1	UPDATE t
—		—
UPDATE t	t2	—
—		—
—	t3	ROLLBACK
—		

Fig. 16.3 Transaction *A* updates an uncommitted change at time *t2*, and loses that update at time *t3*

The temporary update problem.



(b)

T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);	

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

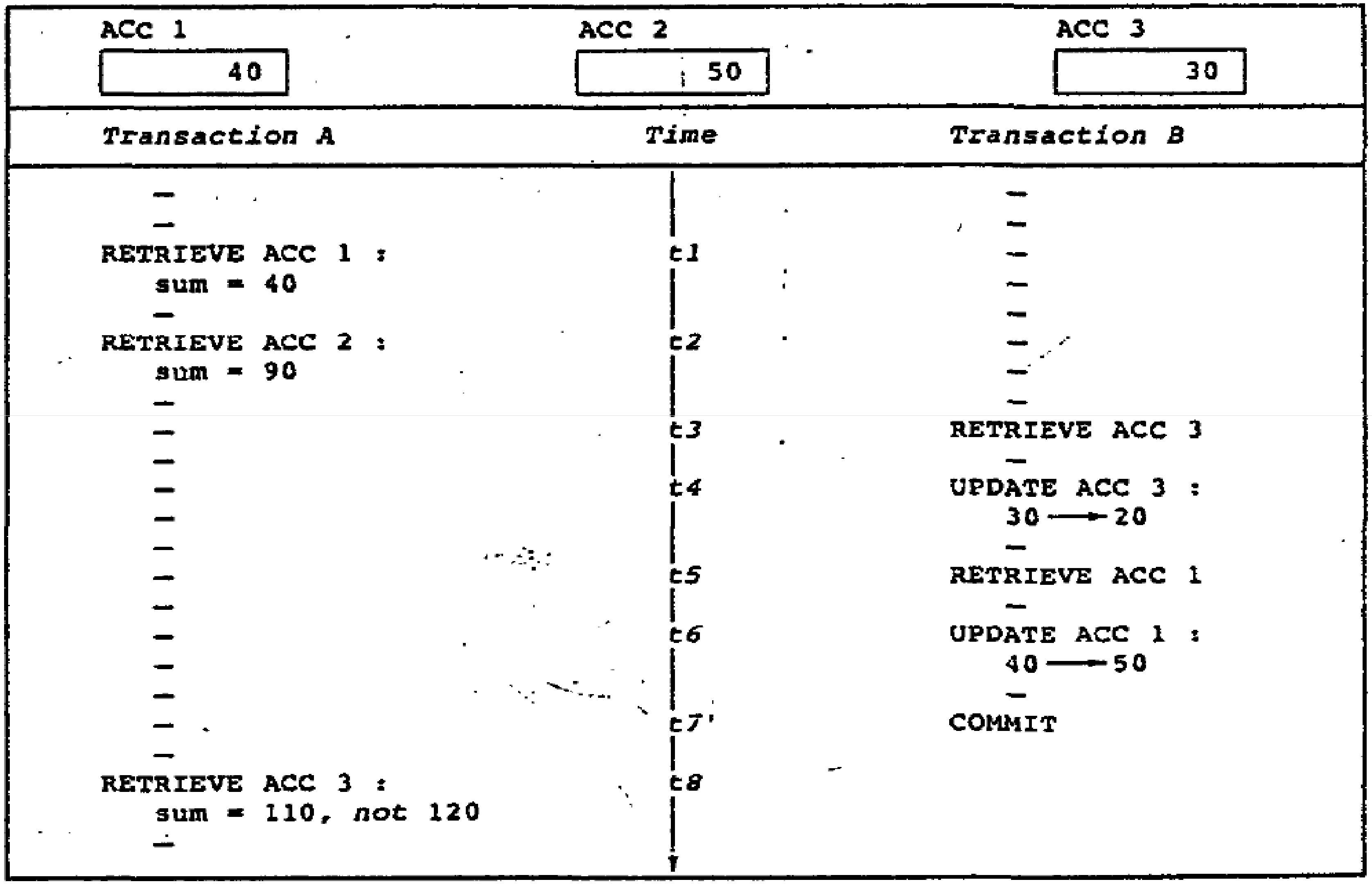
read_item(Y);	

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

- 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.

Inconsistent analysis problem

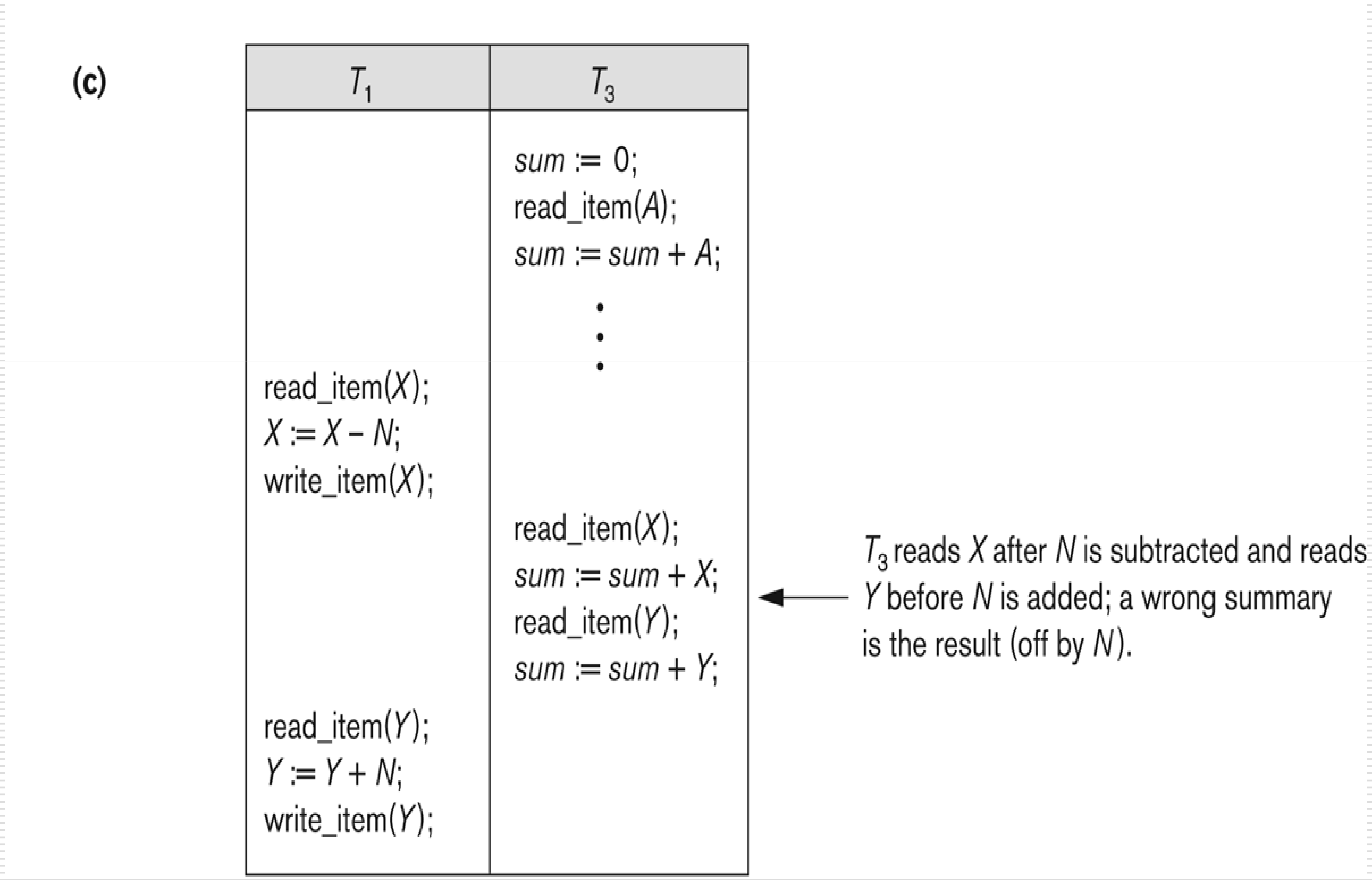


ACC 1	ACC 2	ACC 3
40	50	30

Transaction A	*Time*	*Transaction B*
—		—
—		—
RETRIEVE ACC 1 : sum = 40	*t1*	—
—		—
RETRIEVE ACC 2 : sum = 90	*t2*	—
—		—
—	*t3*	RETRIEVE ACC 3
—		—
—	*t4*	UPDATE ACC 3 : $30 \longrightarrow 20$
—		—
—	*t5*	RETRIEVE ACC 1
—		—
—	*t6*	UPDATE ACC 1 : $40 \longrightarrow 50$
—		—
—	*t7*	COMMIT
—		
RETRIEVE ACC 3 : sum = 110, *not* 120	*t8*	
—		

Fig. 16.4 Transaction *A* performs an inconsistent analysis

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).

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 operation 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.

Introduction to Transaction Processing

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.

Introduction to Transaction Processing

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

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 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 that are executed as part of a transaction.
  - **end_transaction**: This specifies that read and write transaction operations have ended and marks the end limit of transaction execution.
    - At this point it 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.

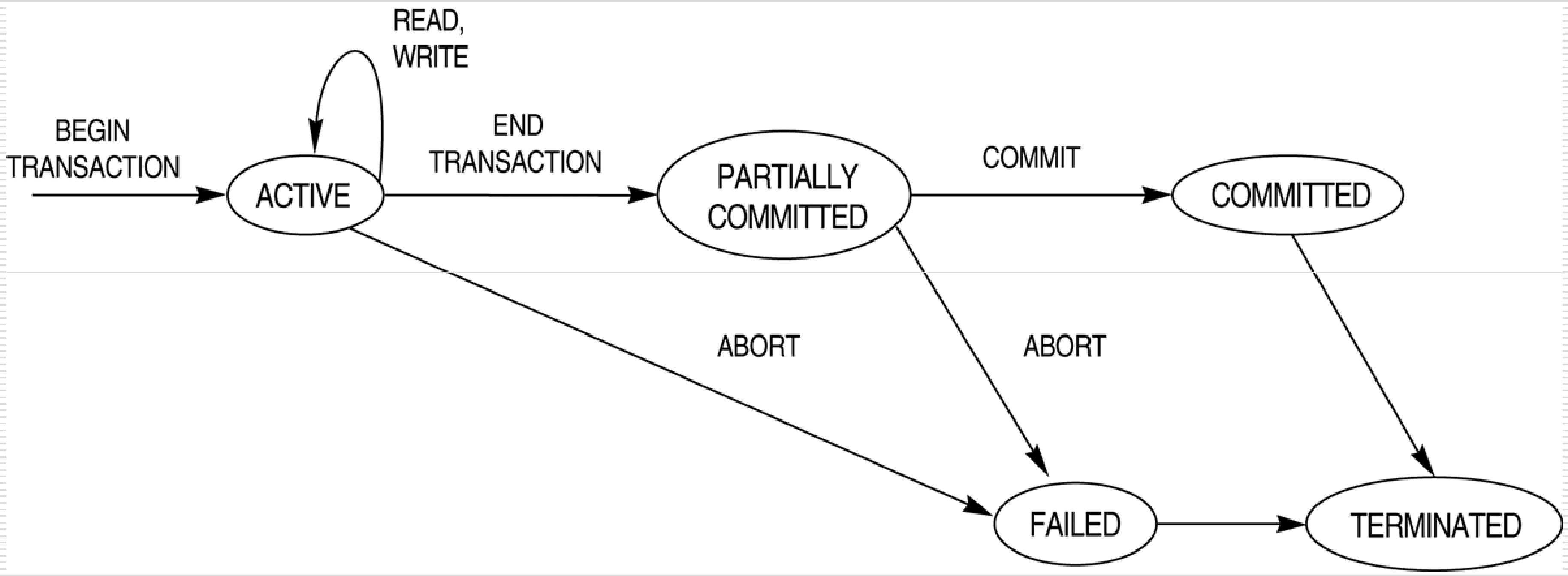
Transaction and System Concepts

- **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.

State transition diagram illustrating the states for transaction execution



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, so it is 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):
  - T in the following discussion refers to a unique **transaction-id** that is generated automatically by the system and is used to identify each transaction:
  - Types of log record:
    - [start_transaction,T]: Records that transaction T has started execution.
    - [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.
    - [read_item,T,X]: Records that transaction T has read the value of database item X.
    - [commit,T]: Records that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
    - [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.
  1. 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.
  2. 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 Commit Point:**
  - 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.

Transaction and System Concepts

Commit Point of a Transaction (cont):

- **Redoing transactions:**
  - Transactions that have written their commit entry in the log must also have recorded all their write operations in the log; otherwise they would not be committed, so their effect on the database can be redone from the log entries. (Notice that the log file must be kept on disk.
  - At the time of a system crash, only the log entries that have been written back to disk are considered in the recovery process because the contents of main memory may be lost.)
- **Force writing a log:**
  - Before a transaction reaches its commit point, any portion of the log that has not been written to the disk yet must now be written to the disk.
  - This process is called force-writing the log file before committing a transaction.

Desirable Properties of Transactions

ACID 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.
- **Isolation**: A transaction should not make its updates visible to other transactions until it is committed; this property, when enforced strictly, solves the temporary update problem and makes cascading rollbacks of transactions unnecessary.
- **Durability or permanency**: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

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

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*)
- **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 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*)
- **Consistency requirement** in above example:
  - the sum of A and B is unchanged by the execution of the transaction

Example of Fund Transfer

- 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

- **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.

Example of transaction

- Transfer £50 from account A to account B

$$\left.\begin{array}{l}\text{Read(A)}\\ \text{A = A - 50}\\ \text{Write(A)}\\ \text{Read(B)}\\ \text{B = B+50}\end{array}\right\}\text{transaction}$$

Write(B)

Atomicity - shouldn’t take money from A without giving it to B

Consistency - money isn’t lost or gained

Isolation - other queries shouldn’t see A or B change until completion

Durability - the money does not go back to A

The Transaction Manager

- The transaction manager enforces the ACID properties
  - It schedules the operations of transactions
  - COMMIT and ROLLBACK are used to ensure atomicity
  - Locks or timestamps are used to ensure consistency and isolation for concurrent transactions
  - A log is kept to ensure durability in the event of system 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 what is known as a transaction schedule (or history).
- A schedule S of n transactions T1, T2, …, Tn:
  - It is an ordering of the operations of the transactions subject to the constraint that, for each transaction Ti that participates in S, the operations of T1 in S must appear in the same order in which they occur in T1.
  - Note, however, that operations from other transactions Tj can be interleaved with the operations of Ti in S.

Characterizing Schedules based on Recoverability

Schedules classified on recoverability:

- Recoverable schedule:
  - One where no transaction needs to be rolled back.
  - 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:
  - One where every transaction reads only the items that are written by committed transactions.

Characterizing Schedules based on Recoverability

Schedules classified on recoverability:

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