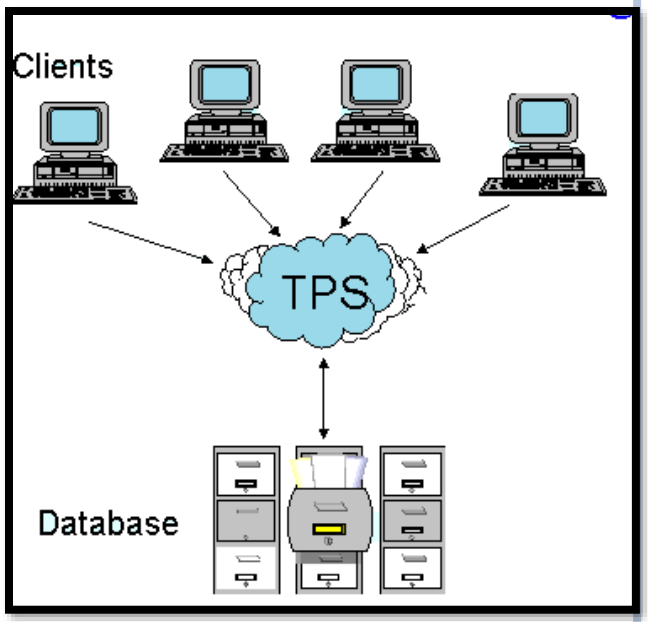
Transactions

Why Transactions?

- *Transaction* is a process involving database queries and/or modification.



- Database systems are normally being accessed by many users or processes at the same time.

- Formed in SQL from single statements or explicit programmer control

ACID Transactions

Atomic	• Whole transaction or none is done.
Consistent	• Database constraints preserved.
Isolated	• It appears to the user as if only one process executes at a time.
Durable	• Effects of a process survive a crash.

Optional: weaker forms of transactions are often supported as well.

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,
    - the **system** should **ensure** that :
      - its **updates** are *not reflected* in the database,
      - else an *inconsistency* will result.

Example of *Fund Transfer*

- Transaction to <u>transfer</u> $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** :
  - the **sum** of **A** and **B** is:
    - **unchanged** <u>by</u> the <u>execution</u> of the transaction.

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*)
- **Isolation requirement** —
  - 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).
  - 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 *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**
      - despite *failures*.

Example: Interacting Processes

Consider a relation **Sells(shop, item, price)**

Let Ahmad sells Sprite for Rs 20 and Pizza slice for Rs 100.

Tania is querying Sells for the highest and lowest price Ahmad charges.

Ahmad decides to stop selling Sprite and Pizza, but to sell Biryani at Rs150.00 per plate.

Customer's Program

- Tania(customer) executes the following two SQL statements called (min) and (max).

(max) SELECT MAX(price) FROM Sells
WHERE shop = `Ahmad"s shop'

(min) SELECT MIN(price) FROM Sells
WHERE shop = `Ahmad"s shop'

Shop Keeper's Program

- At about the same time, Ahmad executes the following steps: (del) and (ins).

(del) DELETE FROM Sells
WHERE shop = 'Ahmad"s shop'

(ins) INSERT INTO Sells
VALUES('Ahmad"s shop', 'Biryani', 150.00)

Interleaving of Statements

- The statement (max) must come before (min), and
- The statement (del) must come before (ins),

- There are no other constraints on the order of these statements.
- Unless we group Tania's and/or Ahmad's statements into transactions.

Example: Strange Interleaving

- Suppose the steps execute in the order (max)(del)(ins)(min).

Ahmad's Prices:	{20,100}	{20,100}	{150}	{150}
Statement:	(max)	(del)	(ins)	(min)
Result:	100			150

- **Tania sees MAX < MIN!**

Another Problem: Rollback

- Suppose Ahmad executes (del)(ins), not as a transaction.
- But after executing these statements, he change his mind and issues a ROLLBACK statement.

- If Tania executes her statements after (ins) but before the rollback, she sees a value, 150, that never existed in the database.

Solution

- If Ahmad executes (del)(ins) as a transaction, its effect cannot be seen by others until the transaction executes COMMIT.
  - If the transaction executes ROLLBACK instead, then its effects can *never* be seen.

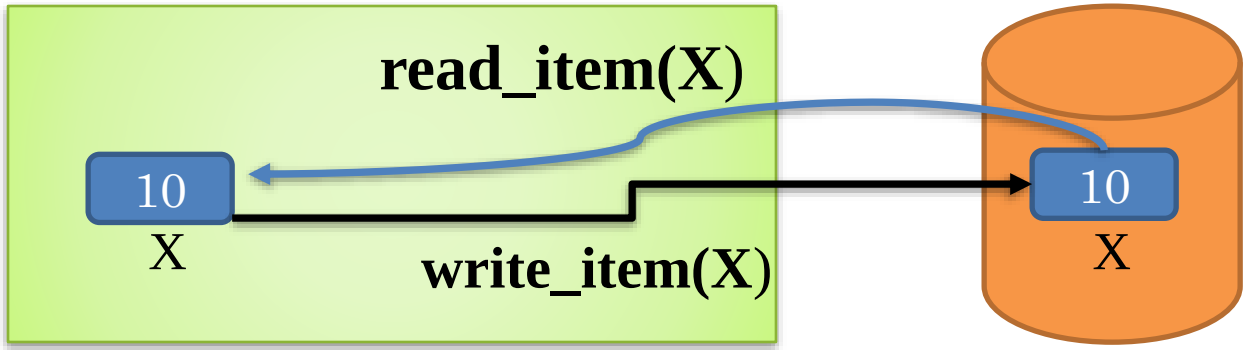
Fixing the Problem by Using Transactions

- Solution: Group Tania's statements (max)(min) into one transaction
  - Now, she cannot see this inconsistency.

- She sees Ahmad's prices at some fixed time.
  - Either before or after he changes prices, or in the middle, but the MAX and MIN are computed from the same prices.

Transaction Processing

- Basic operations in a DB 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.



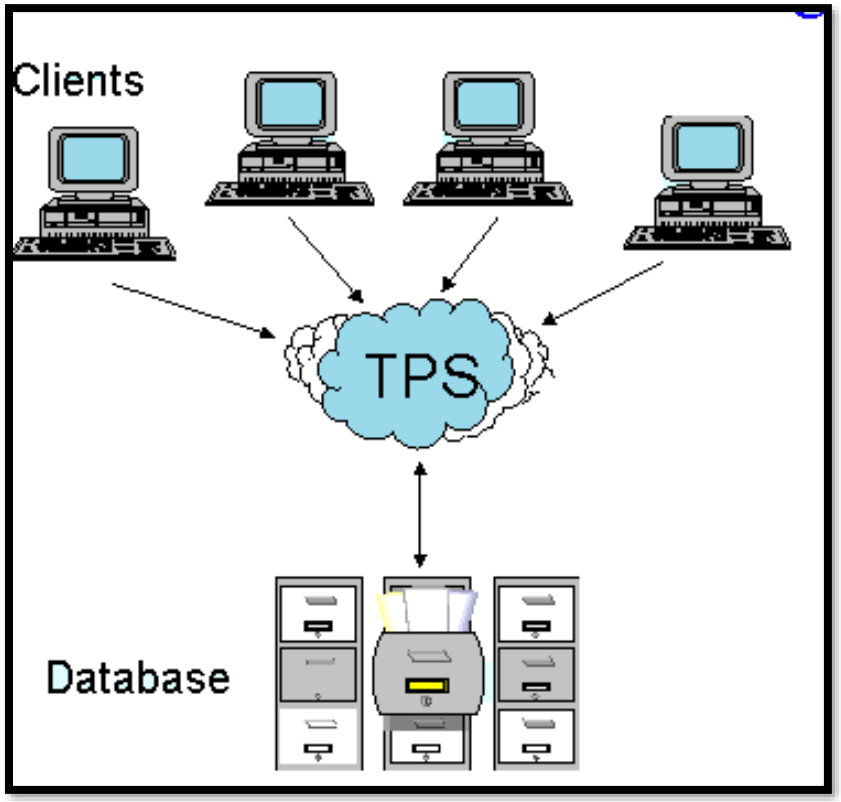
Sample Transactions

(a) T_1

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

(b) T_2

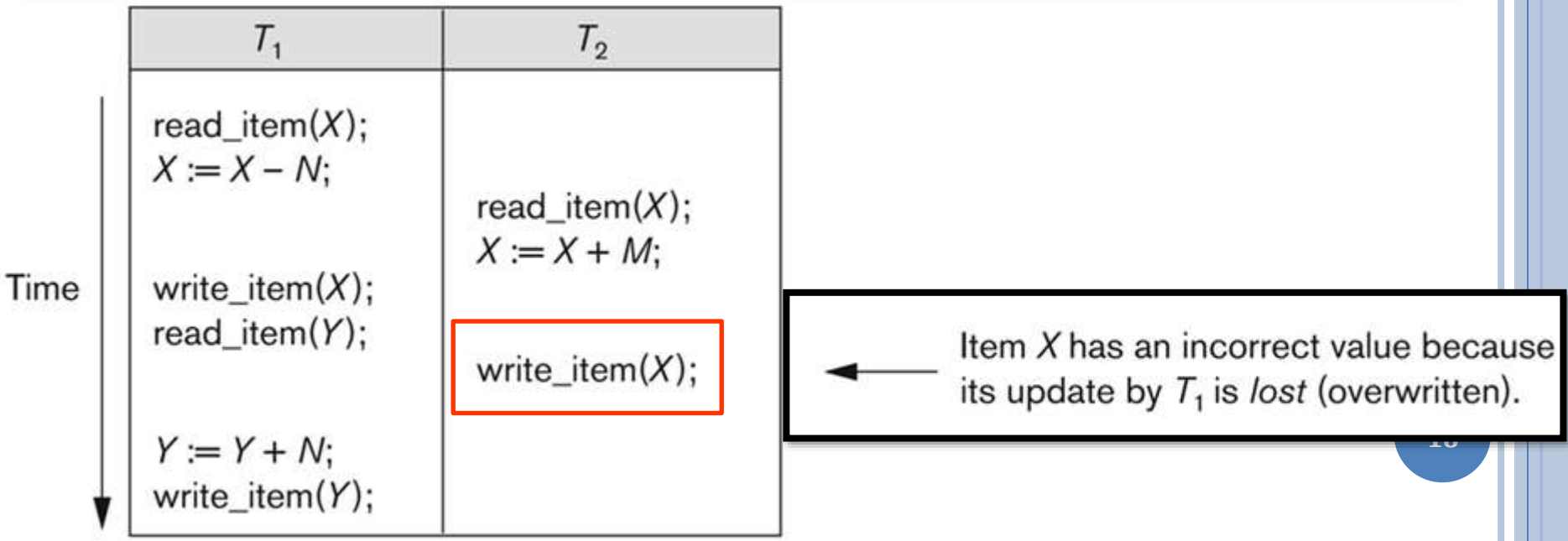
```
read_item (X);
X:=X+M;
write_item (X);
```



Issues in Transaction Processing

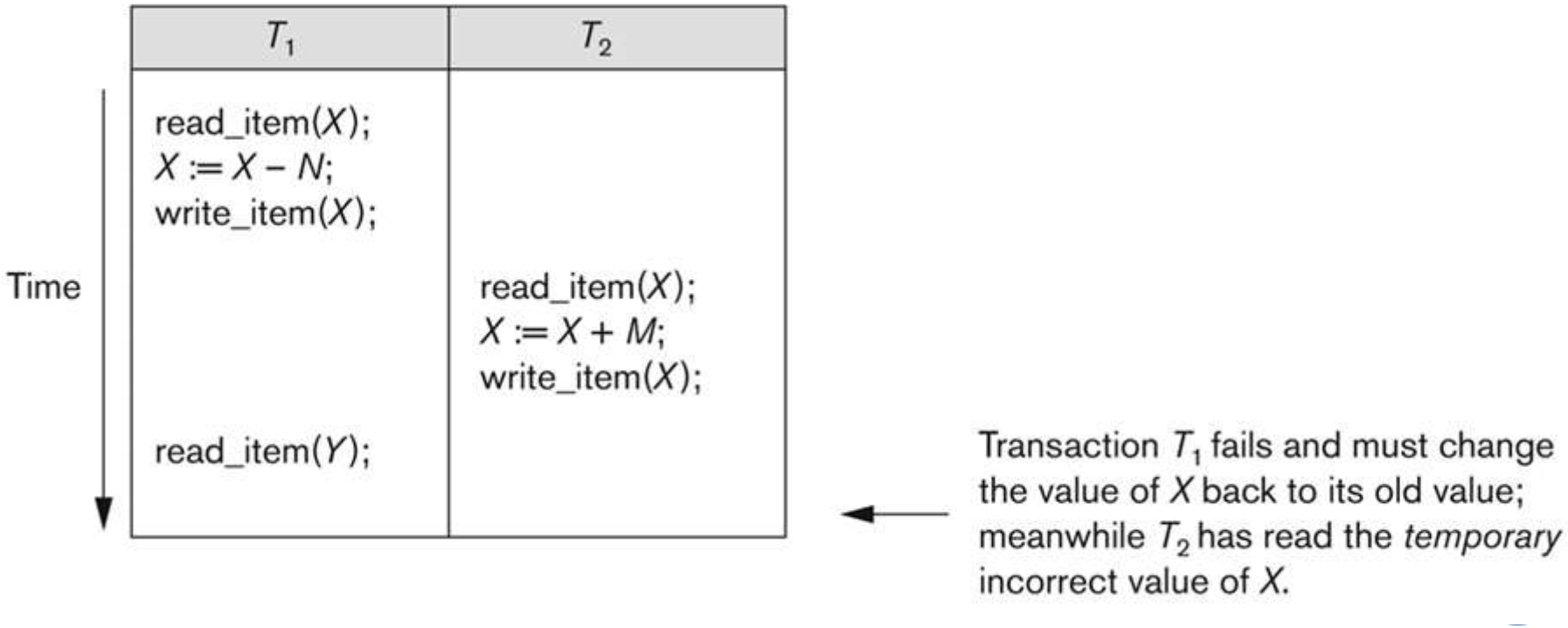
Why Concurrency Control is needed?

- **The Lost Update Problem**
  - Two transactions (that access the same DB items) have their operations interleaved in a way that makes the value of some database item incorrect.



Issues in Transaction Processing

Temporary Update (or Dirty Read) Problem



Issues in Transaction Processing

The Incorrect Summary Problem

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*).

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

Recovery manager keeps track of the following operations

- **Begin_transaction**
- **Read or Write**
- **End_transaction**
- **Commit**
- **Rollback (or abort)**
- **Undo**
- **Redo**

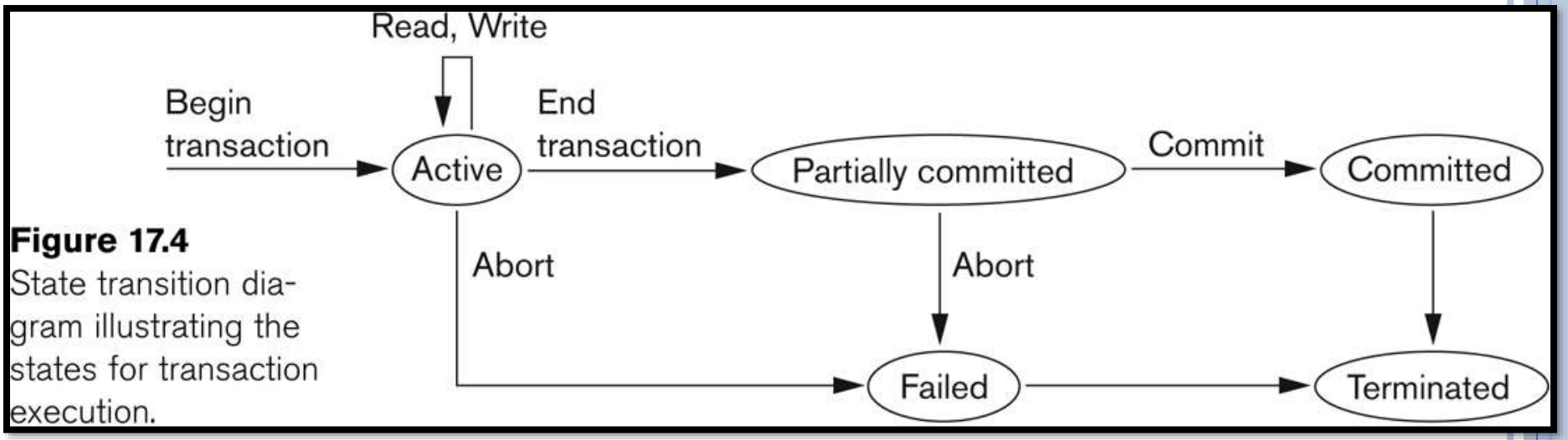
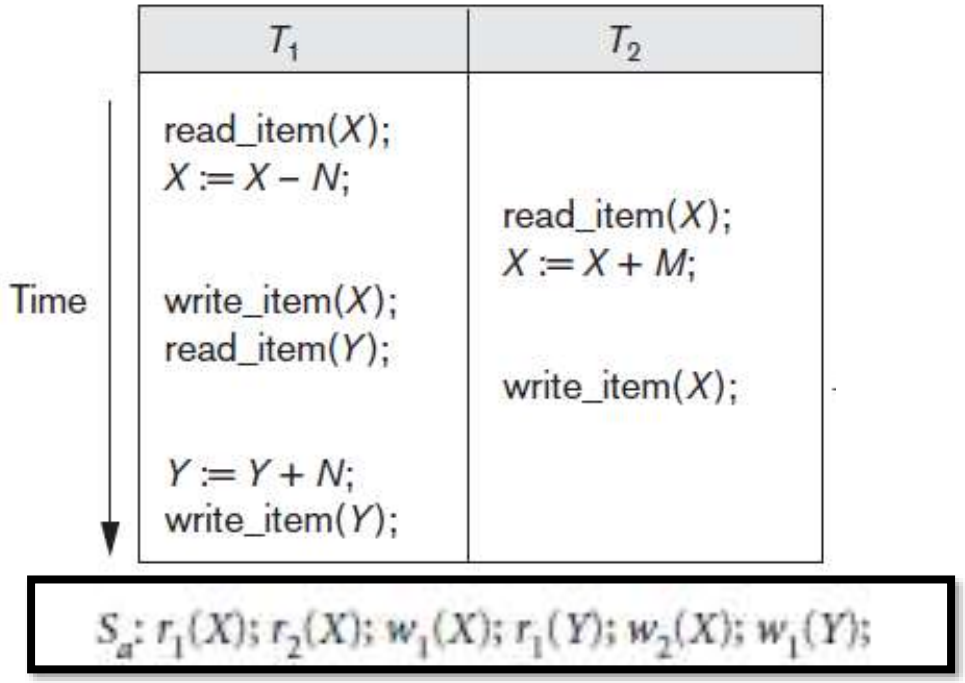


Figure 17.4
State transition diagram illustrating the states for transaction execution.

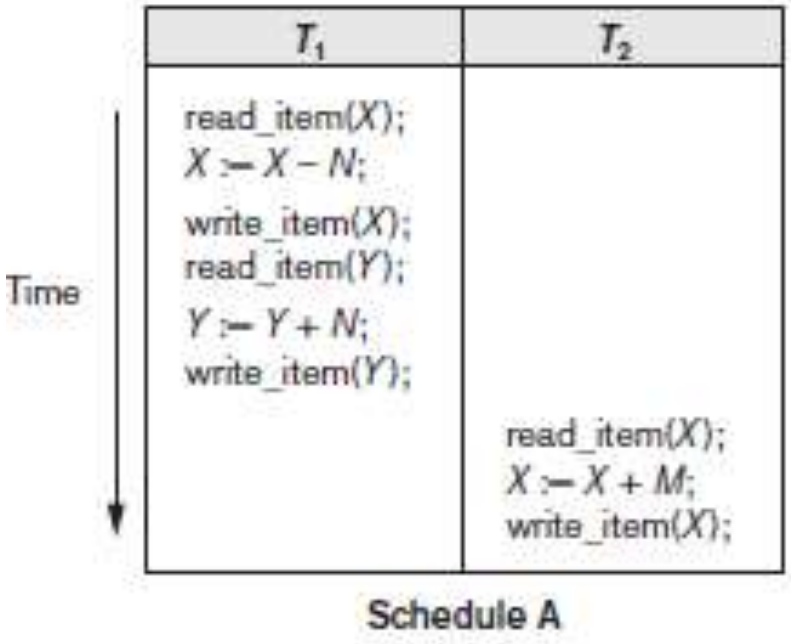
Schedules

- A **schedule** S of n transactions T_1, T_2, ..., T_n is an ordering of the operations of the transactions.
- Operations from different transactions can be interleaved in S.
- The operations of each T_i in S must appear in the same order in which they occur in T_i.



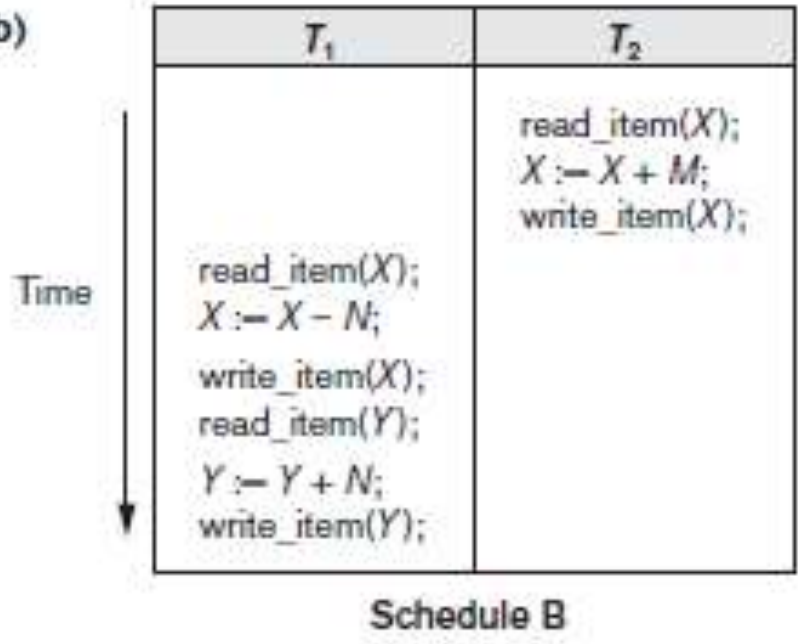
Characterizing Schedules based on Serializability

- **Serial schedule**
  - A schedule S is serial if, for every T in S, all the operations of T are executed consecutively in the schedule.



Schedule A

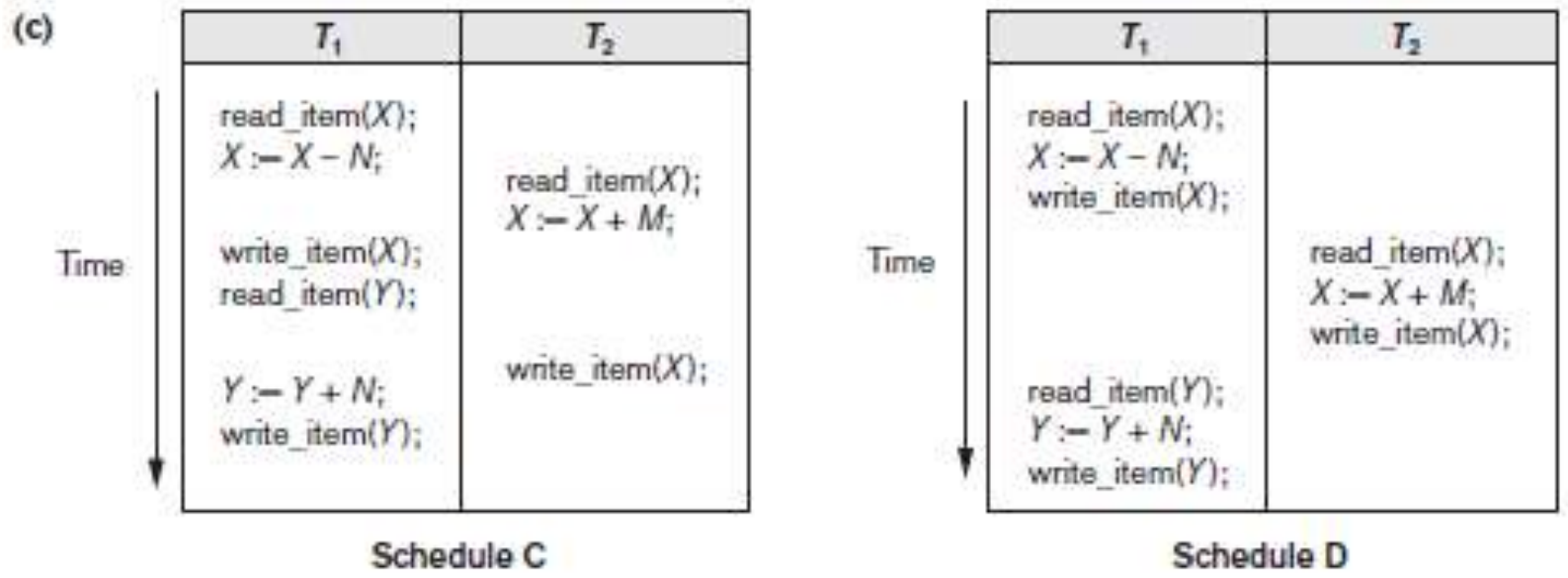
(b)



Schedule B

Characterizing Schedules based on Serializability

- **Non Serial schedule**
  - **Which one is correct ?**



Schedule C gives an erroneous result because of the *lost update problem*
Schedule D gives correct results

Recoverable Schedules

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

Recoverable schedule:

- if a transaction T_2 **reads** a data item *previously* **written** by T_1 .
- then the **commit** of T_1 should appears **before** the **commit** of T_2.

- The following schedule is **not** recoverable **if**
  - T_2 **commits** immediately after the read

T1	T2
r(X)	
w(X)	
	r(X)
r(Y)	

  - If T_1 **abort**, T_2 have done a dirty read .
- Database must ensure that schedules are recoverable.

Characterizing Schedules based on Recoverability

In a recoverable schedule, no committed transaction ever needs to be rolled back

T1	T2
r(X)	
	r(X)
w(X)	
r(Y)	
	w(X)
	commit
w(Y)	
commit	

T1	T2
r(X)	
w(X)	
	r(X)
r(Y)	
	w(X)
	commit
abort	

Which one is 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**)

T1	T2	T3
r(X)		
r(Y)		
w(X)		
	r(X)	
	w(X)	
		r(X)

  - If T_1 *fails*, T_2 and T_3 must also be *rolled back*.
- Can **lead** to the *undoing* of a *significant amount of work*

Characterizing Schedules based on Recoverability

Cascadeless schedule

- A schedule where every transaction reads only the items that are written by committed transactions.
- In other words: *No dirty Read*

Which schedule avoid cascade rollback

S1: r1(X) w1(X) r2(X) r1(Y) w2(X) w1(Y) c1 c2

S2: r1(X) w1(X) r2(X) r1(Y) w2(X) w1(Y) a1 a2

The $r_2(X)$ command in schedules *S1* and *S2* must be postponed until after T_1 has committed (or aborted),

This delays T_2 but ensuring no cascading rollback if T_1 aborts

Characterizing Schedules based on Recoverability

Strict Schedule

A schedule in which a transaction can neither read or write an item X until the last transaction that wrote X has committed.

- Strict schedules simplify the recovery process.
- The process of undoing a **write_item(*X*)** of an aborted transaction is to restore the **before image** (old_value) of *X*.

$$\mathbf{Sf:\ w_1(X,5)\ \ w_2(X,8)\ \ a_1}$$

- *Sf* is cascadeless, but it is not a strict schedule,
  - As it permits T_2 to write item *X* even though T_1 that last wrote *X* had not yet committed (or aborted).

Characterizing Schedules based on Recoverability

Non-Recoverable schedule

- A transaction T_2 **reads** a data item **X** written by T_1 in S and commit before T_1

Recoverable schedule

- If a transaction T_2 **reads** a data item **X** written by T_1 in S and commit after T_1

Cascadeless schedule

- No dirty read

Strict Schedule

- 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

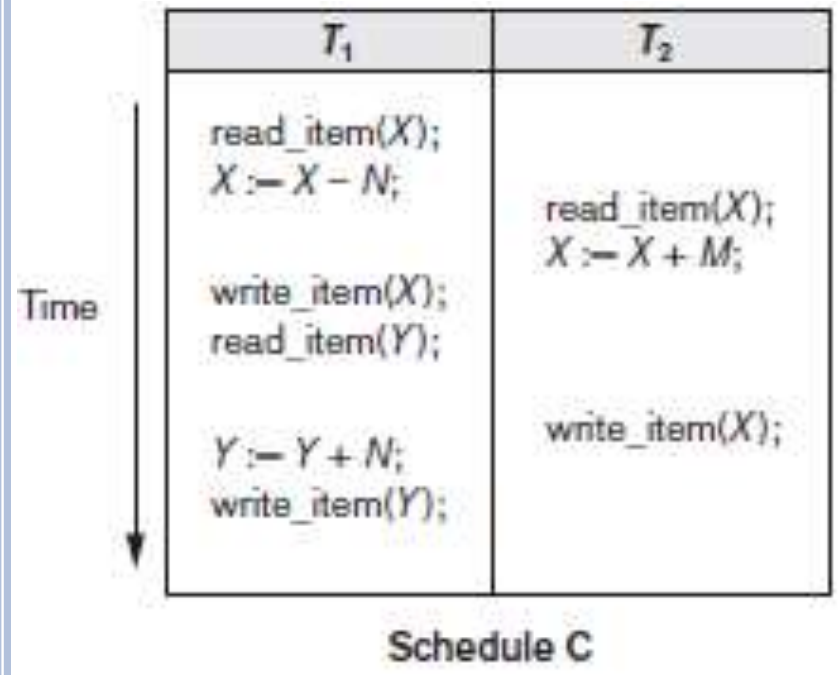
Every **strict** schedule is also **cascadeless**

Every **cascadeless** schedule is also **recoverable**

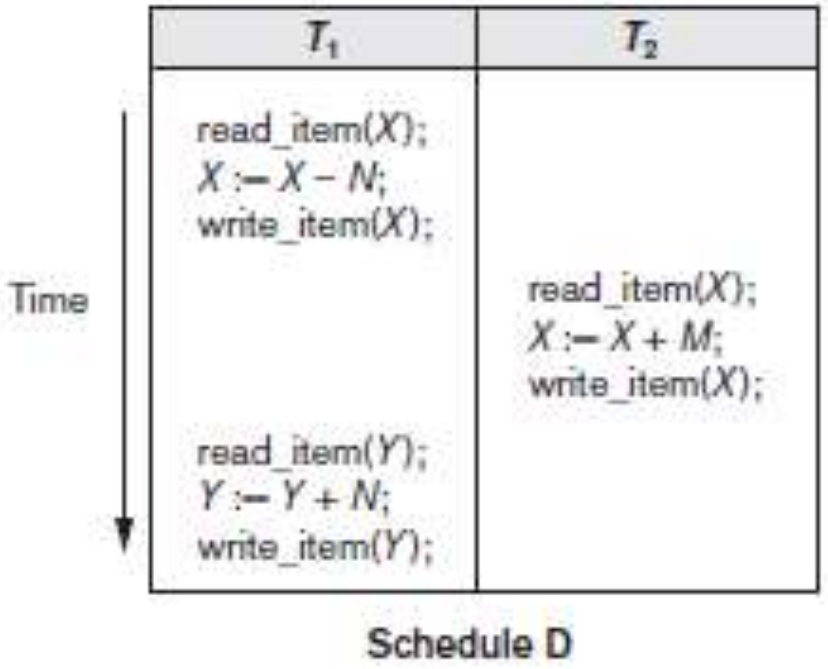
It is **desirable** to **restrict** the schedules to those that are **cascadeless**

Characterizing Schedules based on Serializability

- **Serializable schedule**: A schedule S is serializable if it is equivalent to some serial schedule of the same n transactions.
  - A nonserial schedule *S* is serializable means it is correct,



Schedule C



Schedule D

We would like to determine which of the non-serial schedules *always* give a correct result and which do not .

Schedules- conflict

- Two operations in S are in **conflict** if
  - they belong to ***different transactions***
  - they **access the *same item X*** and
  - *at least one* of the operations is a write_item(*X*).

S: $r_1(x)$ $w_1(X)$ $r_2(X)$ $w_2(x)$ $r_1(y)$ a_1

- **Conflicting operations**
  - $r_1(X)$ and $w_2(X)$
  - $r_2(X)$ and $w_1(X)$
  - $w_1(X)$ and $w_2(X)$

- **Non-conflicting operations**
  - $r_1(X)$ and $r_2(X)$
  - $w_2(X)$ and $w_1(Y)$
  - $r_1(X)$ and $w_1(X)$

Intuitively, two operations are conflicting if changing their order can result in a different outcome.

Types of Conflict: **read-write conflict and write-write conflict**

Characterizing Schedules based on Serializability

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

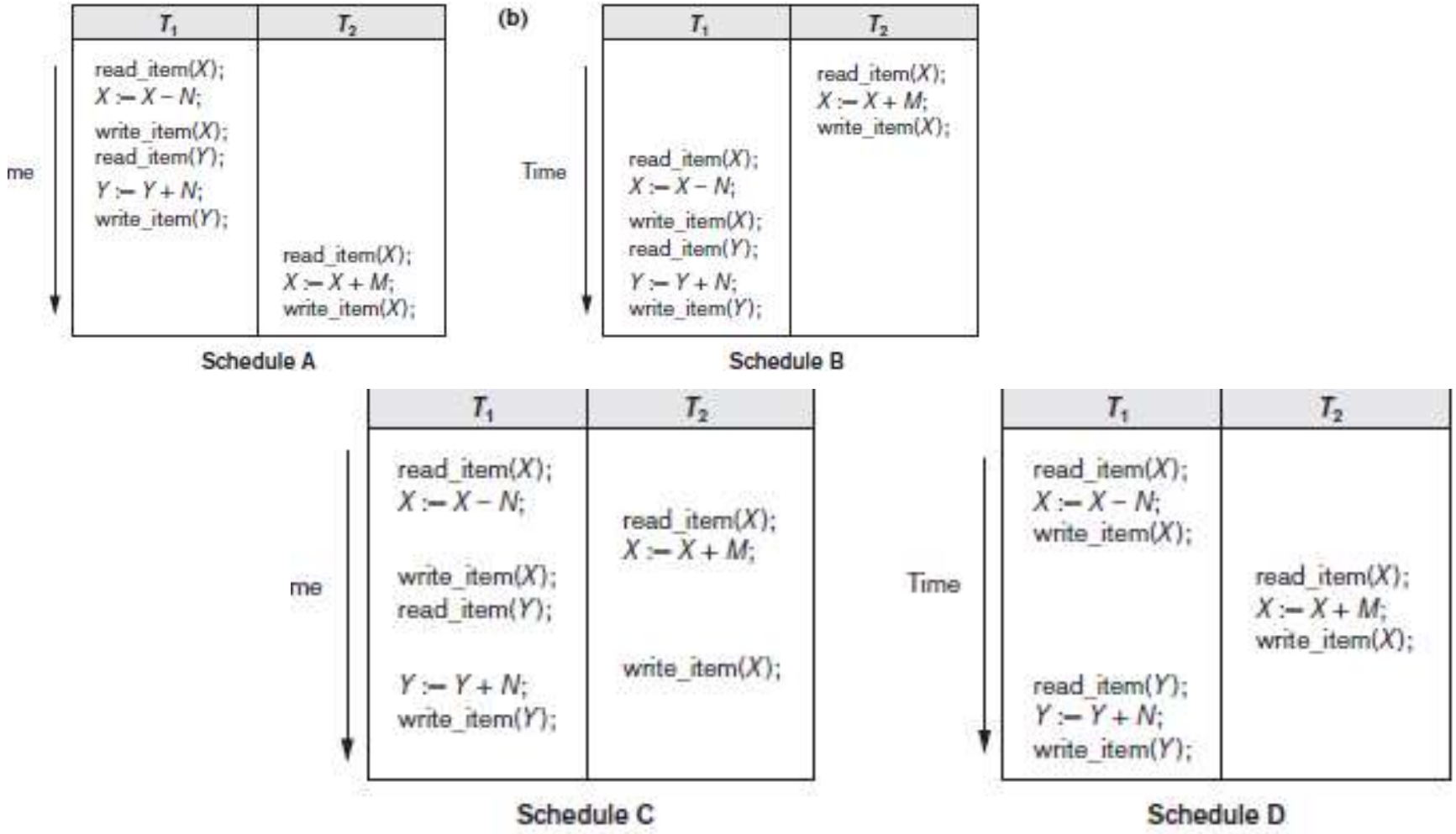
Two operations on the same data items conflict if at least one of them is a write

- $r(X)$ and $w(X)$
- w(X) and r(X)
- $w(X)$ and $w(X)$

- If two conflicting operations are applied in *different orders* in two schedules, the effect can be different

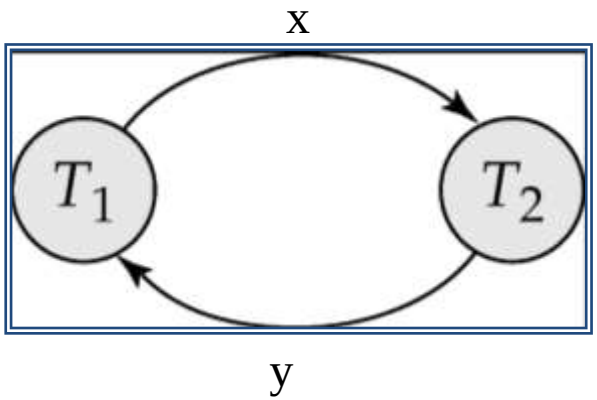
Serializability

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



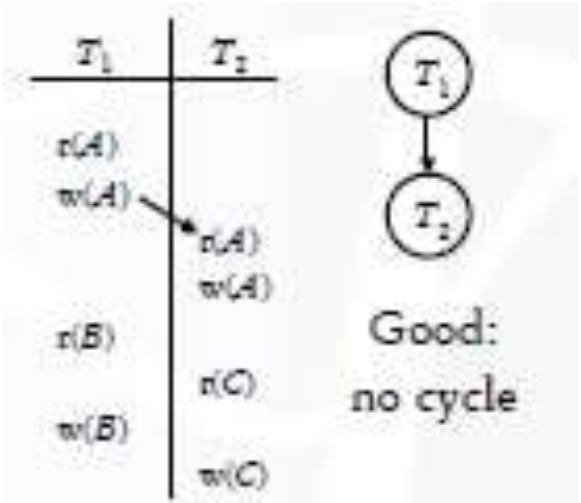
Testing for Serializability

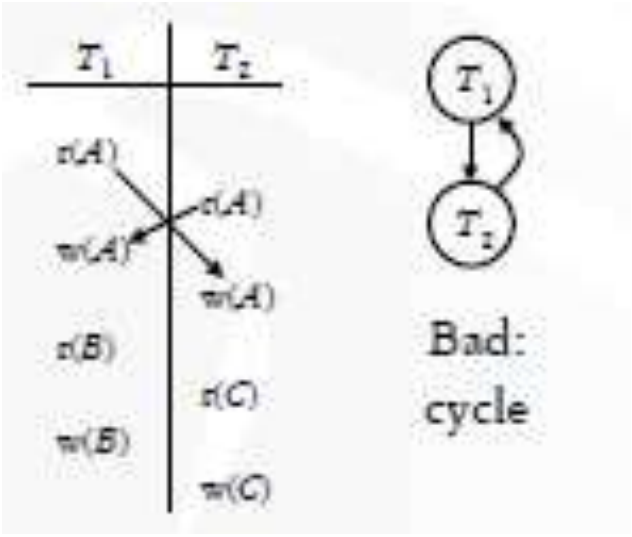
- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- **Precedence graph**
  - a ***direct graph*** where the vertices ***are*** the *transactions* (names).
- Draw an arc from T_i to T_j
  - if the two transaction conflict, and
  - T_i accessed the data item on which the conflict arose **earlier**.
- We may label the arc by the **item** that was **accessed**.



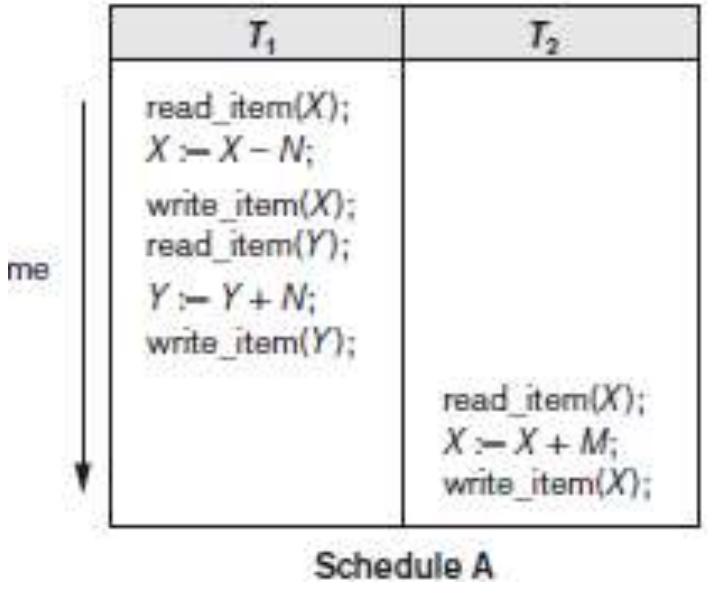
Precedence Graph

- A node for each transaction
- A directed edge from T_i to T_j if an operation of T_i precedes and conflicts with an operation of T_j in the schedule



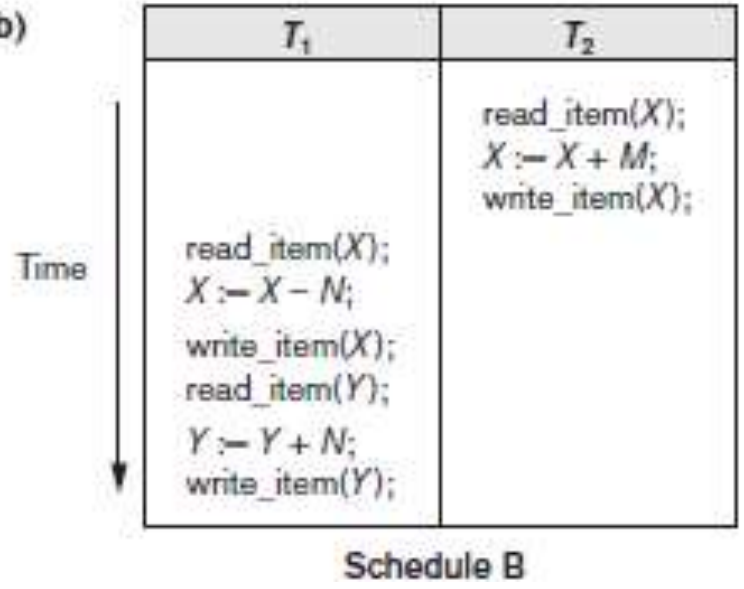


Constructing the Precedence Graphs



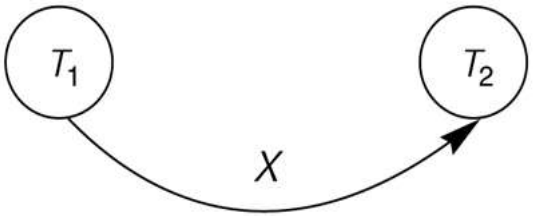
Schedule A

(b)

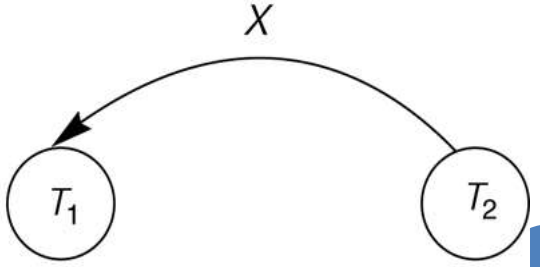


Schedule B

(a)



(b)



Constructing the Precedence Graphs

me

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

Time

T_1	T_2

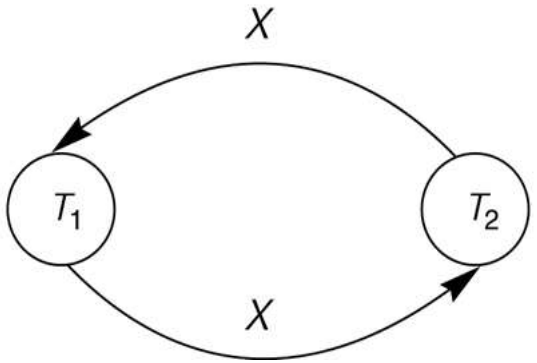
read_item(X);
$X := X - N$;
write_item(X);	

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

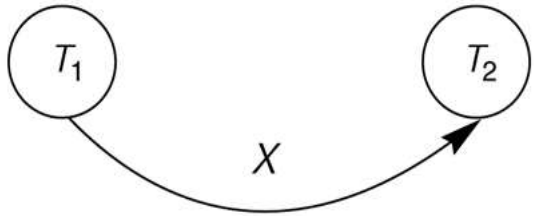
read_item(Y);
$Y := Y + N$;
write_item(Y);	

Schedule D

(c)



(d)



Example of serializability Testing

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);

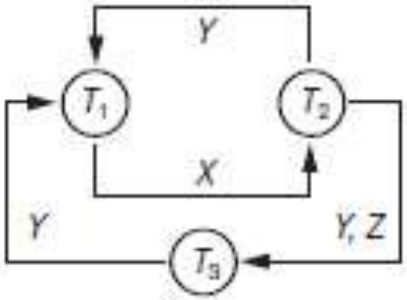
Transaction T_3
read_item(Y);
read_item(Z);
write_item(Y);
write_item(Z);

Example of Serializability Testing

Transaction T_1	Transaction T_2	Transaction T_3
	read_item(Z);	
	read_item(Y);	
	write_item(Y);	
		read_item(Y);
		read_item(Z);
read_item(X);		
write_item(X);		
		write_item(Y);
		write_item(Z);
	read_item(X);	
read_item(Y);		
write_item(Y);		
	write_item(X);	

Time

Schedule E



Equivalent serial schedules

None

Reason

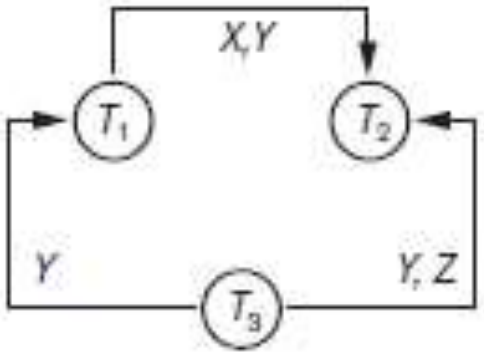
Cycle $X(T_1 \rightarrow T_2), Y(T_2 \rightarrow T_1)$
Cycle $X(T_1 \rightarrow T_2), YZ\ (T_2 \rightarrow T_3), Y(T_3 \rightarrow T_1)$

Example of Serializability Testing

Transaction T_1	Transaction T_2	Transaction T_3
		read_item(Y);
		read_item(Z);
read_item(X);		
write_item(X);		
		write_item(Y);
		write_item(Z);
	read_item(Z);	
read_item(Y);		
write_item(Y);		
	read_item(Y);	
	write_item(Y);	
	read_item(X);	
	write_item(X);	

Time

Schedule F



Equivalent serial schedules

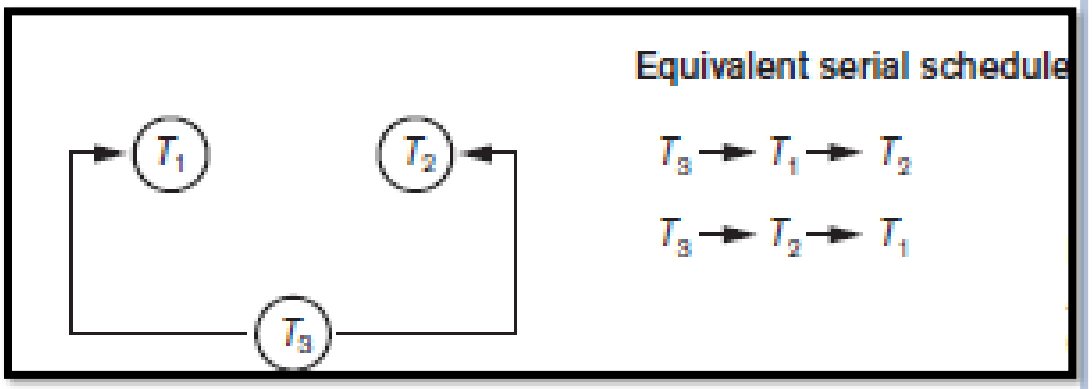
$T_3 \rightarrow T_1 \rightarrow T_2$

Example of Serializability Testing

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);



To find an equivalent serial schedule:

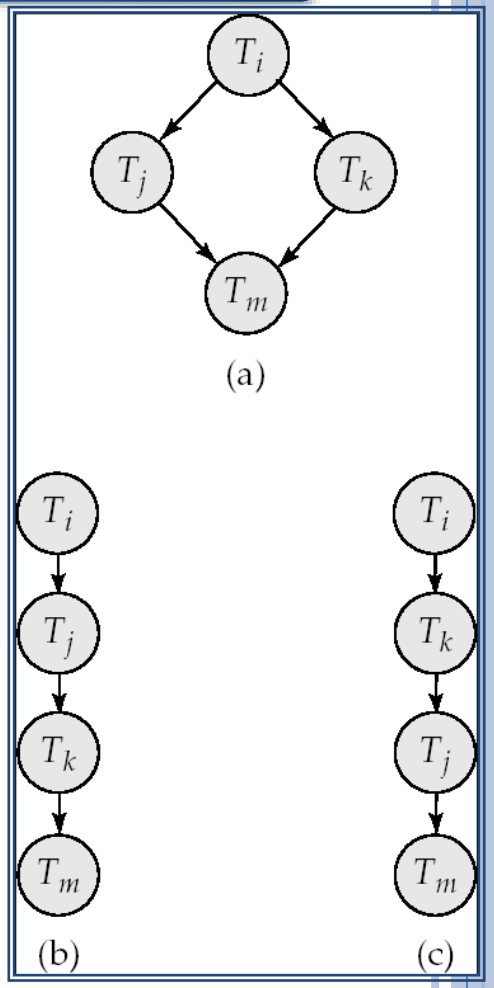
- start with a node that does not have any incoming edges, and then
- make sure that the node order for every edge is not violated.

Test for Conflict Serializability

- A schedule is ***conflict serializable***
  - **if** and only **if** its *precedence graph is* ***acyclic***.

- **Cycle-detection algorithm:**
  - Use ***Depth-first search*** to detect cycle
    - DFS for a connected graph produces a tree.
    - There is a cycle in a graph only if there is a **back edge** present in the graph

- **If** *precedence graph* is **acyclic**, the *serializability order* can be obtained by a ***topological sorting*** of the graph.



A **topological ordering** of a directed graph is a linear ordering of its *vertices* such that for *every directed edge* uv from vertex u to vertex v, u comes before v in the ordering.

T-SQL AND Transactions

SQL has following transaction modes.

- Autocommit transactions
  - Each individual SQL statement = transaction.

- Explicit transactions

  BEGIN TRANSACTION

  [SQL statements]

  COMMIT or ROLLBACK

Transaction Support in TSQL

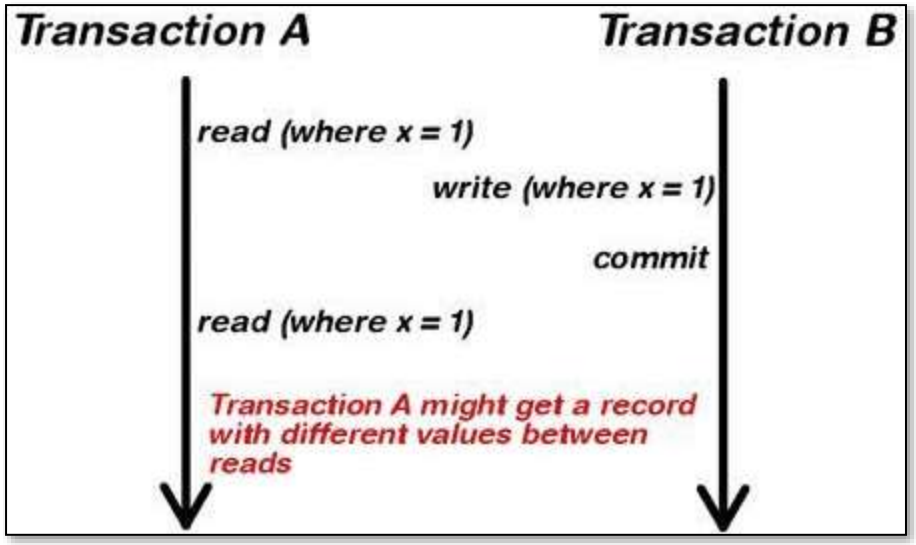
Table 21.1 Possible Violations Based on Isolation Levels as Defined in SQL

	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

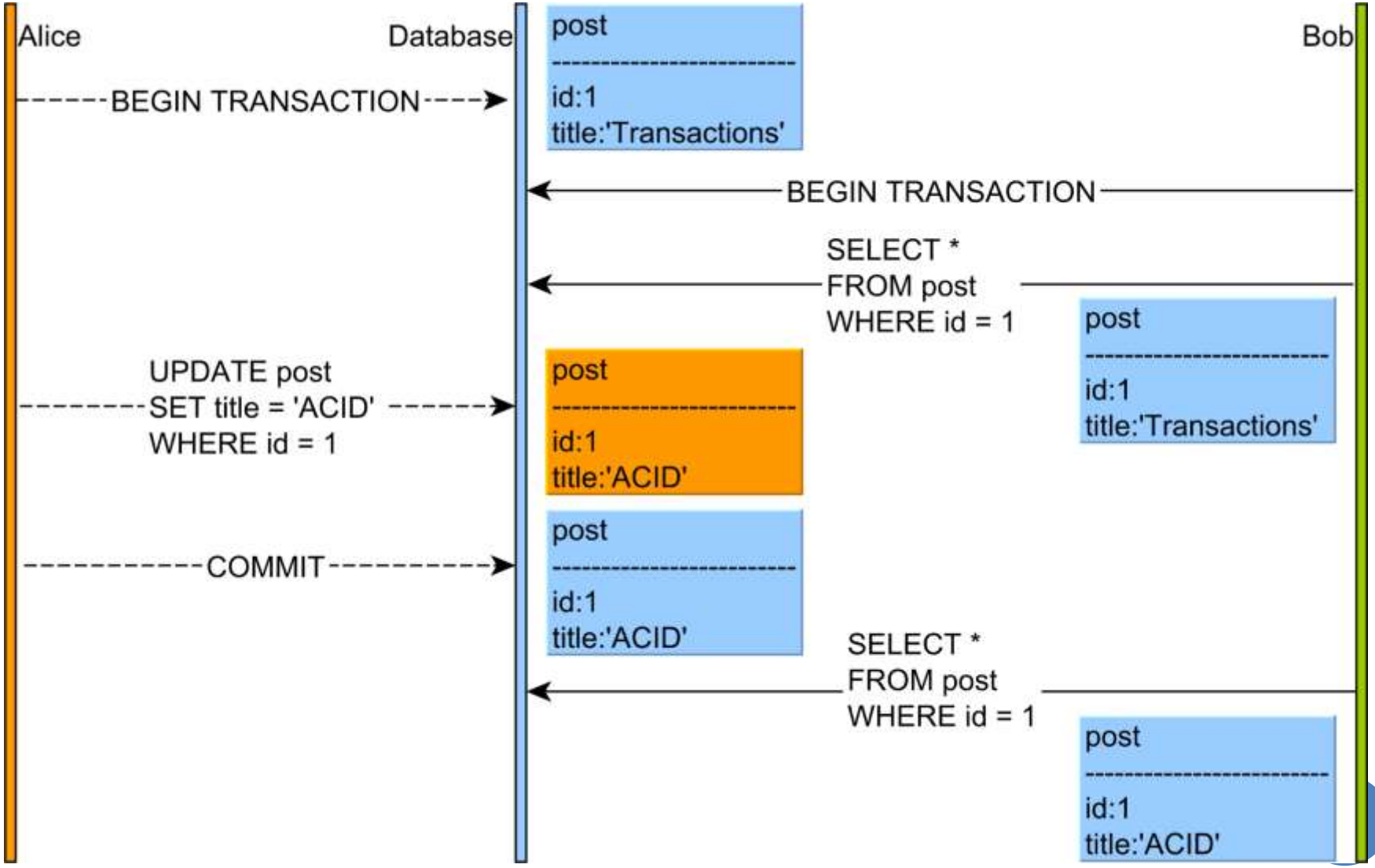
Transaction Support in SQL

Potential problem with lower isolation levels:

- **Dirty Read**
  - Reading a value that was written by a failed transaction.
- **Nonrepeatable Read**
  - A transaction T_i write a new value between multiple reads of transaction T_j.
    - T_j reads a given value X from a table.
    - T_i later updates X and T_j reads that value again, Tj will see a different value.

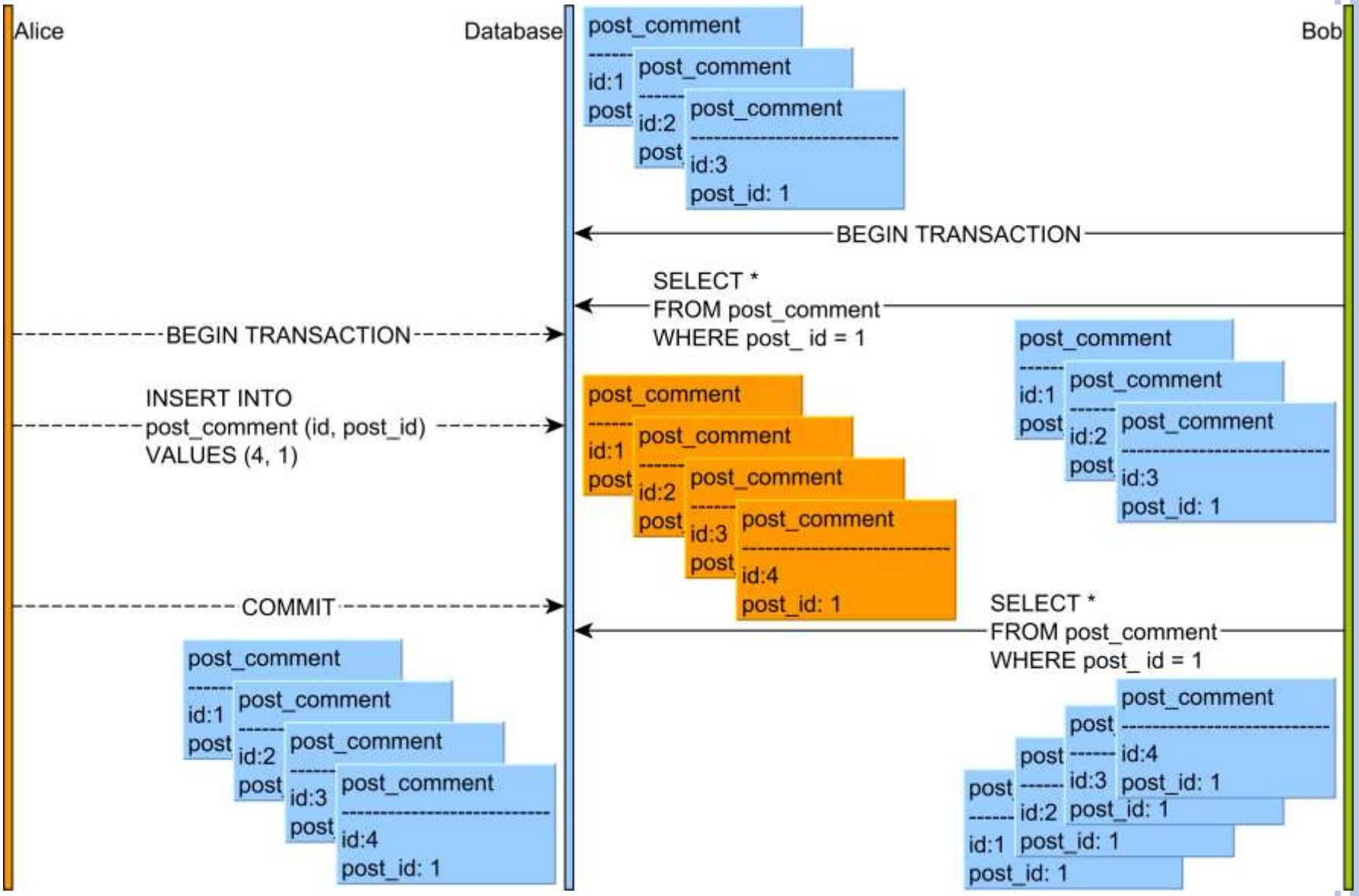


Nonrepeatable Read



https://vladmihalcea.com/non-repeatable-read/

Phantom


Alice
Database
Bob
post_comment
id:1
post_comment
id:2
post_comment
id:3
post_id: 1
BEGIN TRANSACTION
SELECT *
FROM post_comment
WHERE post_ id = 1
BEGIN TRANSACTION
INSERT INTO
post_comment (id, post_id)
VALUES (4, 1)
post_comment
id:4
post_id: 1
COMMIT
SELECT *
FROM post_comment
WHERE post_ id = 1

Transaction Support in SQL

- Potential problem with lower isolation levels (contd.):
  - **Phantoms**
    - New rows being read using the same read with a condition.
    - **Example:**
      - T_1 read a set of rows from a table,
        - on some condition specified in the SQL WHERE clause.
      - 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 TSQL

1. "Dirty reads"

   SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

2. "Committed reads"

   SET TRANSACTION ISOLATION LEVEL READ COMMITTED

3. "Repeatable reads"

   SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

4. Serializable transactions (default):

   SET TRANSACTION ISOLATION LEVEL SERIALIZABLE