Introduction to Transaction Processing Concepts and Theory

What is a Transaction?

- Logical unit of work that must be entirely completed or aborted
- Consists of:
  - SELECT statement
  - Series of related UPDATE statements
  - Series of INSERT statements
  - Combination of SELECT, UPDATE, and INSERT statements
- Specifying the transaction boundaries by Begin and end transaction statements in an application program.
- If the database operations in a transaction do not update the database but only retrieve data, the transaction is called a **read-only transaction.**
- If the database operations in a transaction update the database after the retrieve data, the transaction is called a **read-write transaction.**

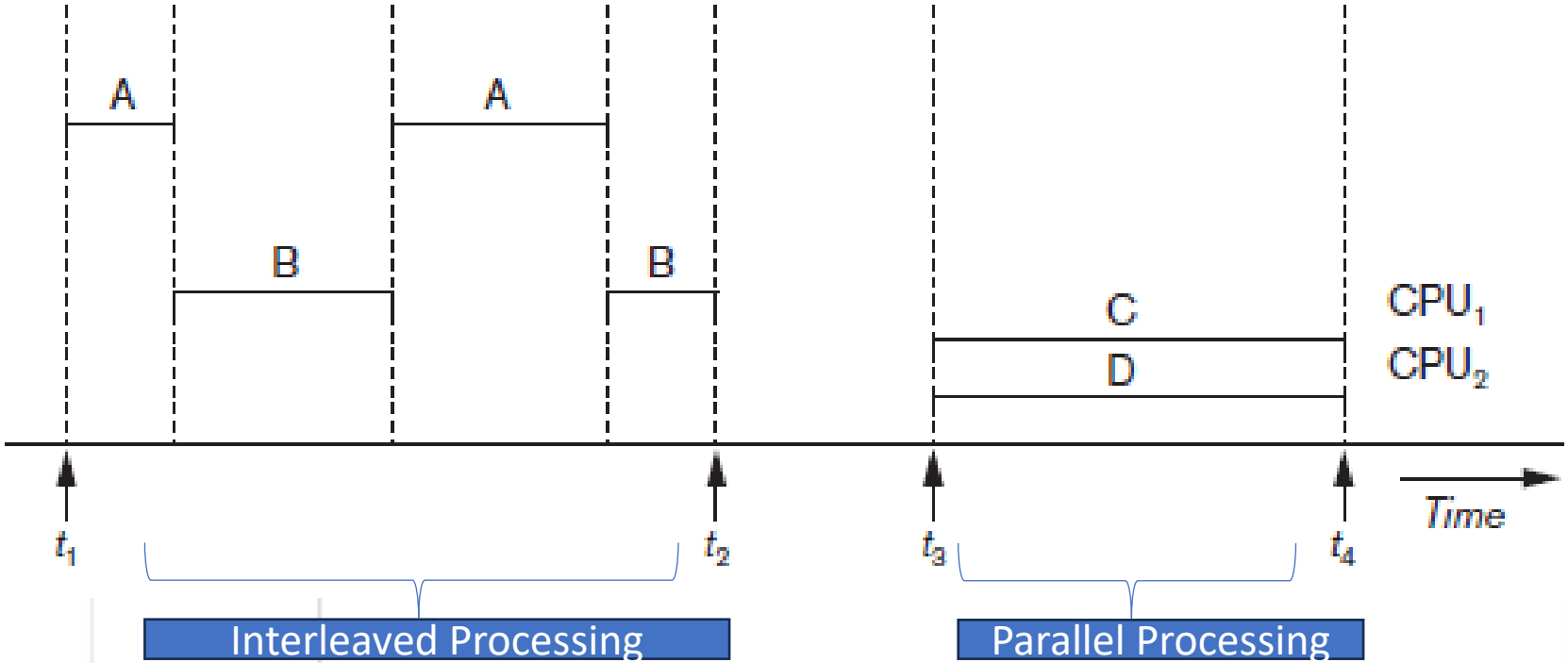
What is a Transaction?

- **Consistent database state**: All data integrity constraints are satisfied
  - Must begin with the database in a known consistent state to ensure consistency
- Formed by two or more database requests
  - **Database requests**: Equivalent of a single SQL statement in an application program or transaction

Database Users

- Single-user DBMS
  - At most one user at a time can use the system
  - Example: home computer
- Multiuser DBMS
  - Many users can access the system (database) concurrently
  - Example: airline reservations system
- Multiprogramming
  - Allows operating system to execute multiple processes concurrently
  - Executes commands from one process, then suspends that process and executes commands from another process, etc.
  - Concurrent execution process is interleaved.

• If computer have multiple hardware processors(CPUs), **parallel processing** is possible.


A
A
B
B
C
D
CPU1
CPU2
t1
t2
t3
t4
Time
Interleaved Processing
Parallel Processing

Database Models

- Represents the transaction processing concept.
- Database represented as collection of named data items.
  - Data item
    - Record
    - Disk block
    - Attribute value of a record
- Size of a data item called its granularity
- Transaction processing concepts independent of item granularity.

Basic Database operations

- read_item(X)
  - Reads a database item named X into a program variable named X
  - Process includes finding the address of the disk block, and copying to and from a memory buffer
- write_item(X)
  - Writes the value of program variable X into the database item named X
  - Process includes finding the address of the disk block, copying to and from a memory buffer, and storing the updated disk block back to disk

Read and Write Operations

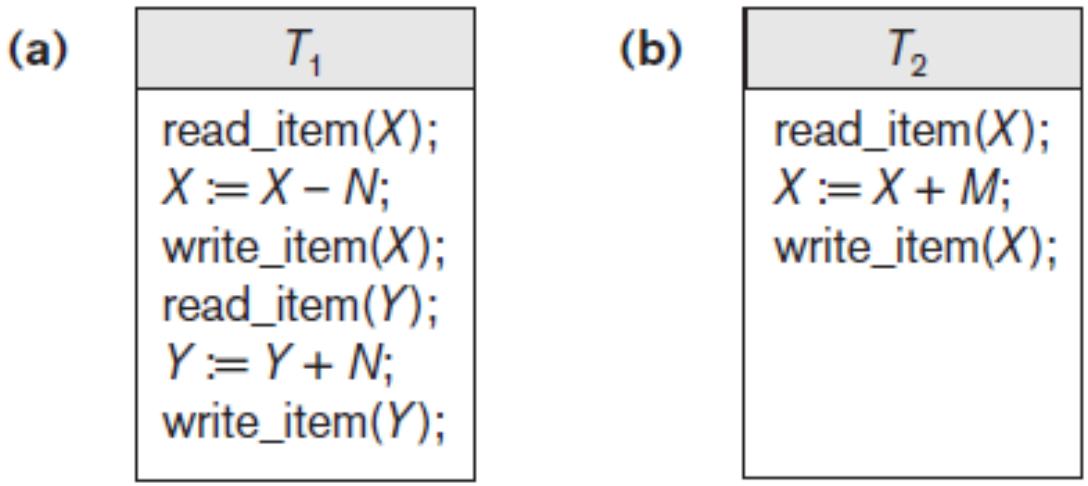
read_item(X)

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). The size of the buffer is the same as the disk block size.
3. Copy item *X* from the buffer to the program variable named *X*.

write_item(X)

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 disk block from the buffer back to disk (either immediately or at some later point in time).

Read and Write Operations



DBMS Buffers

- DBMS will maintain several main memory data buffers in the database cache
- When buffers are occupied, a buffer replacement policy is used to choose which buffer will be replaced
  - Example policy: least recently used

Concurrency Control

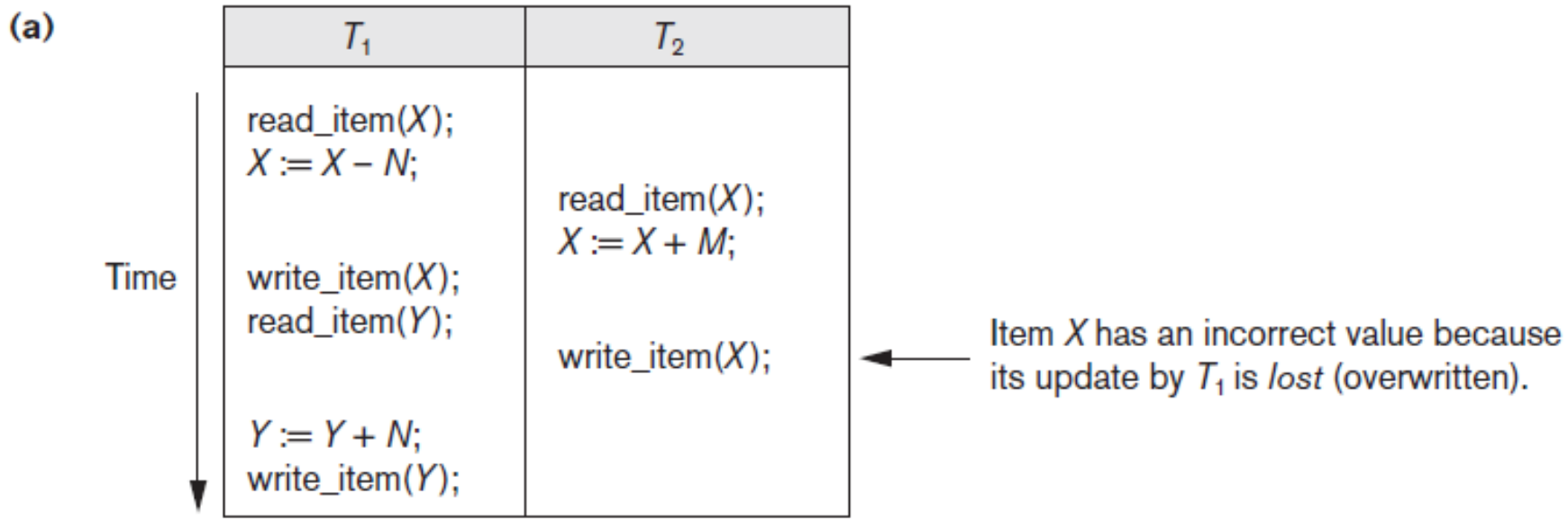
- Transactions submitted by various users may execute concurrently
  - Access and update the same database items
  - Some form of concurrency control is needed

Concurrency Control with respect to Airline Reservation Example

- Each record includes the *number of reserved seats* on that flight as a *named (uniquely identifiable) data item*, among other information.
- Transaction *T*1 that *transfers N* reservations from one flight whose number of reserved seats is stored in the database item named *X* to another flight whose number of reserved seats is stored in the database item named *Y*.
- Transaction *T*2 that just *reserves M* seats on the first flight (*X*) referenced in transaction *T*1.
- When a database access program is written, it has the flight number, the flight date, and the number of seats to be booked as parameters; hence, the same program can be used to execute *many different transactions*, each with a different flight number, date, and number of seats to be booked.
- For concurrency control purposes, a transaction is a *particular execution* of a program on a specific date, flight, and number of seats.
- The transactions *T*1 and *T*2 are *specific executions* of the programs that refer to the specific flights whose numbers of seats are stored in data items *X* and *Y* in the database.

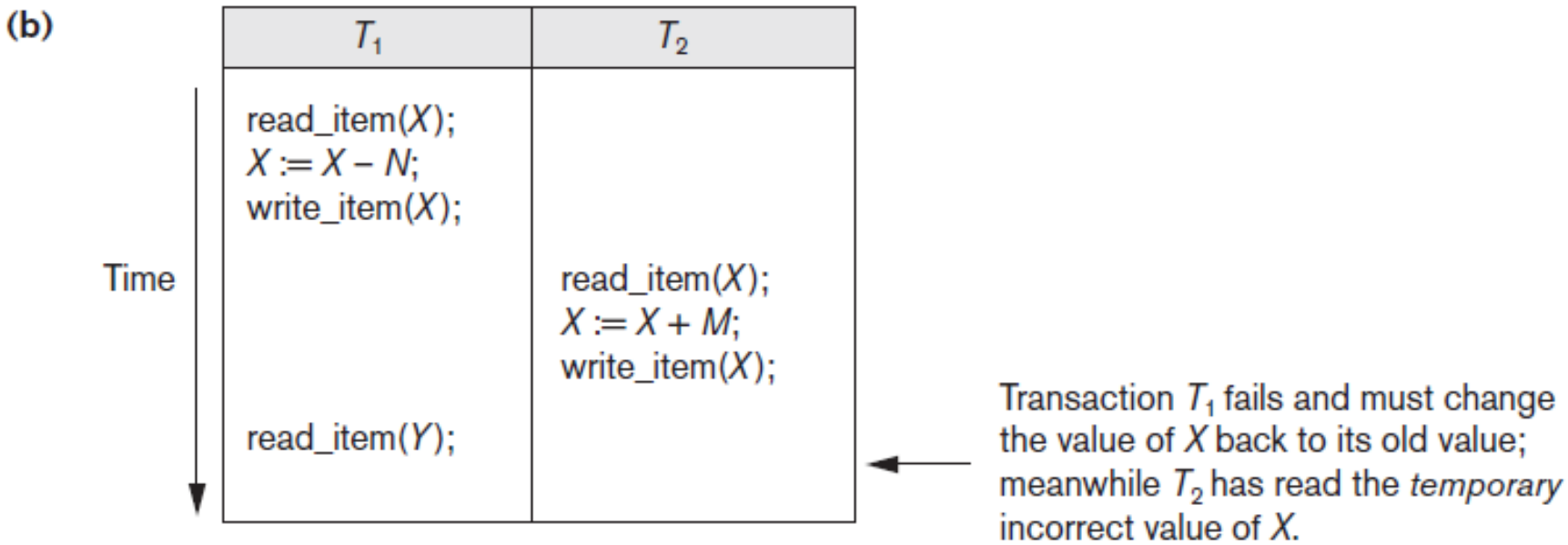
The lost update problem

- Occurs when two transactions that access the same database items have operations interleaved
- Results in incorrect value of some database items



The Temporary Update Problem

- This problem occurs when one transaction updates a database item and then the transaction fails for some reason.
- Meanwhile, the updated item is accessed (read) by another transaction before it is changed back (or rolled back) to its original value.



The Incorrect Summary Problem

- If one transaction is calculating an aggregate summary function on several database items while other transactions are updating some of these items, the aggregate function may calculate some values before they are updated and others after they are updated.

(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).

The Unrepeatable Read Problem

- A transaction *T* reads the same item twice and the item is changed by another transaction *T'* between the two reads. Hence, *T* receives *different values* for its two reads of the same item.

If during an airline reservation transaction, a customer inquires about seat availability on several flights. When the customer decides on a particular flight, the transaction then reads the number of seats on that flight a second time before completing the reservation, and it may end up reading a different value for the item.

Problems in Concurrency Control

Lost update

- Occurs in two concurrent transactions when:
  - Same data element is updated
  - One of the updates is lost

Uncommitted data

- Occurs when:
  - Two transactions are executed concurrently
  - First transaction is rolled back after the second transaction has already accessed uncommitted data

Inconsistent retrievals

- Occurs when a transaction accesses data before and after one or more other transactions finish working with such data

Why Recovery is Needed

- Whenever a transaction is submitted to a DBMS for execution, the system is responsible for making sure that either all the operations in the transaction are completed successfully and their effect is recorded permanently in the database, or that the transaction does not have any effect on the database or any other transactions.
- Committed transaction
  - Effect recorded permanently in the database
- Aborted transaction
  - Does not affect the database

Types of transaction failures

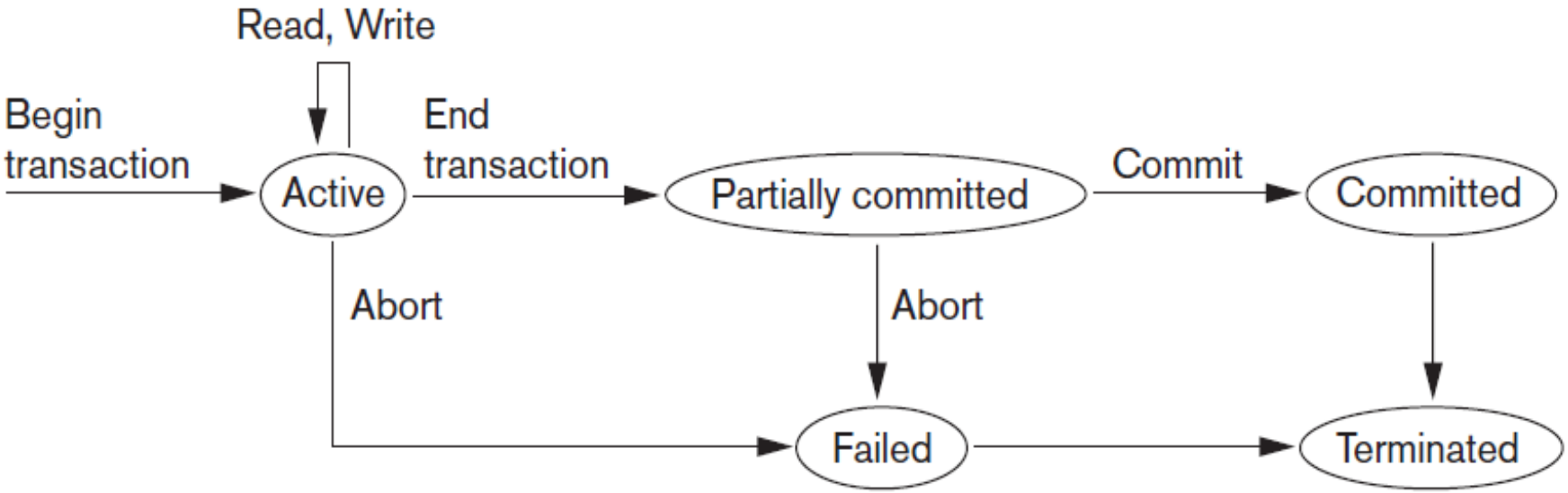
1. Computer failure (system crash)
2. Transaction or system error
3. Local errors or exception conditions detected by the transaction
4. Concurrency control enforcement
5. Disk failure
6. Physical problems or catastrophes

- System must keep sufficient information to recover quickly from the failure
  - Disk failure or other catastrophes have long recovery times

Transaction states and System Concepts

- A transaction is an atomic unit of work that should either be completed in its entirety or not done at all.
- For recovery purposes, the system needs to keep track of when each transaction starts, terminates, and commits, or aborts
- The recovery manager of the DBMS needs to keep track of the following operations:
  - BEGIN_TRANSACTION
  - READ or WRITE
  - END_TRANSACTION
  - COMMIT_TRANSACTION
  - ROLLBACK (or ABORT)

Transaction states and System Concepts



The System Log

- System log keeps track of transaction operations
- Sequential, append-only file
- Not affected by failure (except disk or catastrophic failure)
- Log buffer
  - Main memory buffer
  - When full, appended to end of log file on disk
- Log file is backed up periodically.
- Undo and redo operations based on log possible

Log Records

1. **[start_transaction, *T*].** Indicates that transaction *T* has started execution.
2. **[write_item, *T*, *X*, *old_value*, *new_value*].** Indicates that transaction *T* has changed the value of database item *X* from *old_value* to *new_value*.
3. **[read_item, *T*, *X*].** Indicates that transaction *T* has read the value of database item ***X***.
4. **[commit, *T*].** Indicates that transaction *T* has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
5. **[abort, *T*].** Indicates that transaction *T* has been aborted.

Commit Point of a Transaction

- Occurs when all operations that access the database have completed successfully
  - And effect of operations recorded in the log
- Transaction writes a commit record into the log
  - If system failure occurs, can search for transactions with recorded start_transaction but no commit record
- Force-writing the log buffer to disk
  - Writing log buffer to disk before transaction reaches commit point

DBMS-Specific Buffer Replacement Policies

- In a Database Management System (DBMS), **buffer replacement policies** determine how data pages are loaded into and removed from the **buffer cache** (also known as the buffer pool).
- The buffer cache is a part of memory used to temporarily store pages (data blocks) read from the database's disk storage. Since accessing data from memory is much faster than accessing it from disk, the DBMS uses the buffer to improve performance by minimizing direct disk I/O (input/output) operations.
- The DBMS cache will hold the disk pages that contain information currently being processed in main memory buffers.

DBMS-Specific Buffer Replacement Policies Methods

Domain separation method

- Selects buffers to be replaced when all are full
- Each domain handles one type of disk pages
  - Index pages
  - Data file pages
  - Log file pages
- Number of available buffers for each domain is predetermined

Hot set method

- Useful in queries that scan a set of pages repeatedly
- Does not replace the set in the buffers until processing is completed

The DBMIN method

- Predetermines the pattern of page references for each algorithm for a particular type of database operation
- Calculates locality set using query locality set model (QLSM)

Desirable Properties of Transactions

Atomicity	All operations of a transaction must be completed •If not, the transaction is aborted
Consistency	Permanence of database's consistent state
Isolation	Data used during transaction cannot be used by second transaction until the first is completed
Durability	Ensures that once transactions are committed, they cannot be undone or lost
Serializability	Ensures that the schedule for the concurrent execution of several transactions should yield consistent results

Desirable Properties of Transactions

- Levels of isolation
  - Level 0 isolation does not overwrite the dirty reads of higher-level transactions
  - Level 1 isolation has no lost updates
  - Level 2 isolation has no lost updates and no dirty reads
  - Level 3 (true) isolation has repeatable reads
    - In addition to level 2 properties

Characterizing Schedules Based on Recoverability

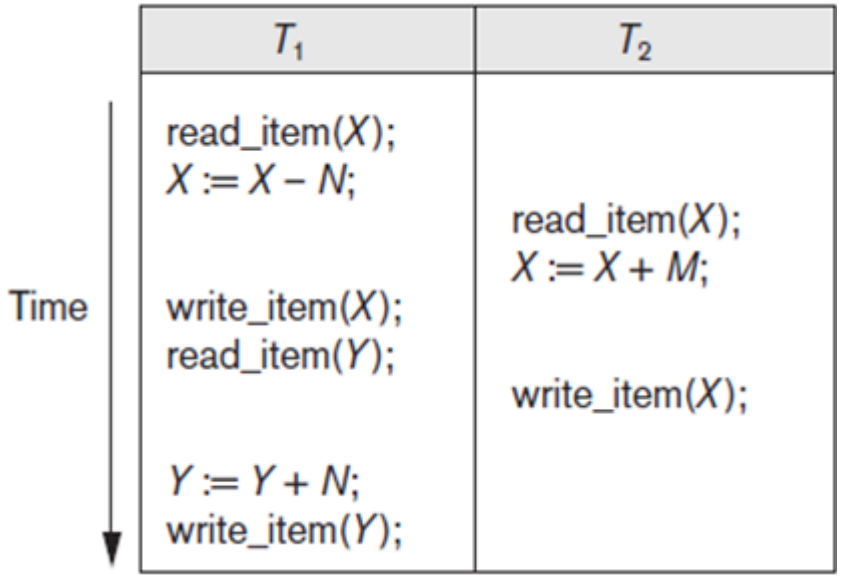
- When transactions are executing concurrently in an interleaved fashion, then the order of execution of operations from all the various transactions is known as a **schedule** (or **history**).
- A **schedule** (or **history**) *S* of *n* transactions T_1, T_2, … , T_n is an ordering of the operations of the transactions.
- Operations from different transactions can be interleaved in the schedule *S*.
- For each transaction T_i that participates in the schedule *S*, the operations of T_i in *S* must appear in the same order in which they occur in T_i.
- The order of operations in *S* is a *total ordering*, meaning *that for any two operations* in the schedule, one must occur before the other.
- In concurrency and recovery, we usually deal with two operations Read_operations and write_operations.

Schedules(histories) of transaction Conflicting Conditions

- Two conflicting operations in a schedule if satisfy all three of the following conditions are said to be in conflict:
  - Operations belong to different transactions
  - Operations access the same item X
  - At least one of the operations is a write_item(X)
- Two operations conflict if changing their order results in a different outcome.

Schedules(histories) of transaction

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



Conflict Conditions:

1. $r_1(X)$ and $w_2(X)$
2. $r_2(X)$ and $w_1(X)$
3. $w_1(X)$ and $w_2(X)$

Non conflict Conditions:

1. $r_1(X)$ and $r_2(X)$
2. $w_2(X)$ and $w_1(Y)$
3. $r_1(X)$ and $w_1(X)$

Order Changing example

1. $r_1(X)$ and $w_2(X)$ ***to*** $w_2(X)$ and $r_1(X)$ **(read –write conflict)**
2. $w_1(X)$ and $w_2(X)$ ***to*** $w_2(X)$ and $w_1(X)$ **(write-write conflict)**

Schedules(histories) of transaction (Complete Schedule)

- The operations in *S* are exactly those operations in *T*1, *T*2, … , *Tn*, 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 *Ti*, their relative order of appearance in *S* is the same as their order of appearance in *Ti*.
- For any two conflicting operations, one of the two must occur before the other in the schedule.

Characterizing Schedules Based on Recoverability

- Recoverable schedules
  - Once a transaction *T* is committed, it should *never* be necessary to roll back *T*. This ensures that the durability property of transactions is not violated.
- Nonrecoverable schedules should not be permitted by the DBMS
- No committed transaction ever needs to be rolled back
- Cascading rollback may occur in some recoverable schedules
  - Uncommitted transaction may need to be rolled back

Characterizing Schedules Based on Recoverability (recoverable and nonrecoverable transaction

Recoverable schedules

- once a transaction *T* is committed, it should *never* be necessary to roll back *T*.

Step	Transaction T_1	Transaction T_2	Explanation
1	`write(X)`		T_1 writes a new value to X.
2		`read(X)`	T_2 reads the value of X written by T_1.
3	`commit`		T_1 commits.
4		`commit`	T_2 commits.

Nonrecoverable Schedules

- A schedule where a committed transaction may have to be rolled back during recovery is called **nonrecoverable** and hence should not be permitted by the DBMS.

Step	Transaction T_1	Transaction T_2	Explanation
1	`write(X)`		T_1 writes a new value to X.
2		`read(X)`	T_2 reads the value of X written by T_1.
3		`commit`	T_2 commits before T_1 commits.
4	`abort`		T_1 aborts.

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

T1	T2
Read_item(X); Write_item(X)	
	Read_item(X);
Read_item(Y);	
	Write_item(x)
	Commit;
abort	

S_a: $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; c_2;$w_1(Y)$;c_1; **R after serialization**

Time

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

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

T1	T2
Read_item(X); Write_item(X)	
	Read_item(X);
Read_item(Y);	
	Write_item(x)
Commit	
	Commit

S_e: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; a_1; a_2; **R**

T1	T2
Read_item(X); Write_item(X)	
	Read_item(X);
Read_item(Y);	
	Write_item(x)
abort	
	abort

Characterizing Schedules Based on Recoverability (Cascading rollback)

S_e: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; a_1; a_2; **R**

Step	Transaction T_1	Transaction T_2	Explanation
1	`write(X)`		T_1 writes a new value to X.
2		`read(X)`	T_2 reads the value of X written by T_1.
3	`abort`		T_1 aborts, rolling back its changes.
4		`rollback`	T_2 must roll back because it read X from T_1.

- *Uncommitted* transaction must be rolled back because it read an item from a transaction that failed.

Characterizing Schedules Based on Recoverability (Cascade less rollback)

- Cascading rollback can be time consuming.

Step	Transaction T_1	Transaction T_2	Explanation
1	`write(X)`		T_1 writes a new value to X.
2	`commit`		T_1 commits its changes.
3		`read(X)`	T_2 reads the value of X written by T_1.
4		`commit`	T_2 commits safely.

Characterizing Schedules Based on Recoverability (strict Schedule)

- A transaction can read or write an item X **only after the last transaction that wrote X has committed or aborted**.
- No transaction is allowed to read or write an item written by an uncommitted transaction.

Step	Transaction T_1	Transaction T_2	Explanation
1	`write(X)`		T_1 writes a new value to X.
2	`commit`		T_1 commits its changes.
3		`read(X)` or `write(X)`	T_2 reads or writes X only after T_1 commits.

Difference between Recoverable, cascade less and strict schedule

Aspect	Recoverable Schedule	Cascade less Schedule	Strict Schedule
Read Condition	Transactions can read uncommitted data but commit only after the writer has committed.	Transactions can read only data written by **committed transactions**.	Transactions can read data only after the writer has **committed or aborted**.
Write Condition	No restriction on writes.	Transactions can write data even if the previous writer has not committed.	Transactions can write data only after the previous writer has **committed or aborted**.
Risk of Dirty Writes	**Possible**: A transaction can overwrite uncommitted data.	**Possible**: A transaction may overwrite uncommitted data.	**Not Possible**: No overwriting of uncommitted data.
Risk of Cascading Rollbacks	**Possible**: If a transaction aborts, dependent transactions must roll back	**Eliminated**: Because transactions read only committed data.	**Eliminated**: Because transactions read only committed data.
Isolation Level	Weaker than cascadeless and strict schedules.	Weaker compared to strict schedules.	Stronger than cascadeless schedules.
Performance	High, as there are fewer delays.	Fewer delays than strict schedules, potentially higher performance.	More delays due to stricter restrictions, potentially lower performance.

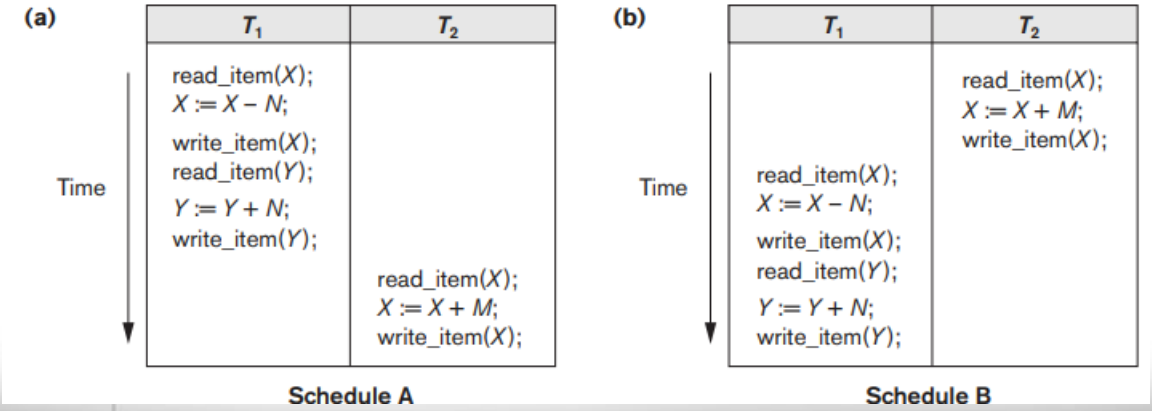
It is important to note that any strict schedule is also cascade less, and any cascade less schedule is also recoverable.

Characterizing Schedules Based on Serializability

- Characterize the types of schedules that are always considered to be correct when concurrent transactions are executing.
- Schedules are known as serializable schedules.
- The concept of serializability of schedules is used to identify which schedules are correct when transaction executions have interleaving of their operations in the schedules.

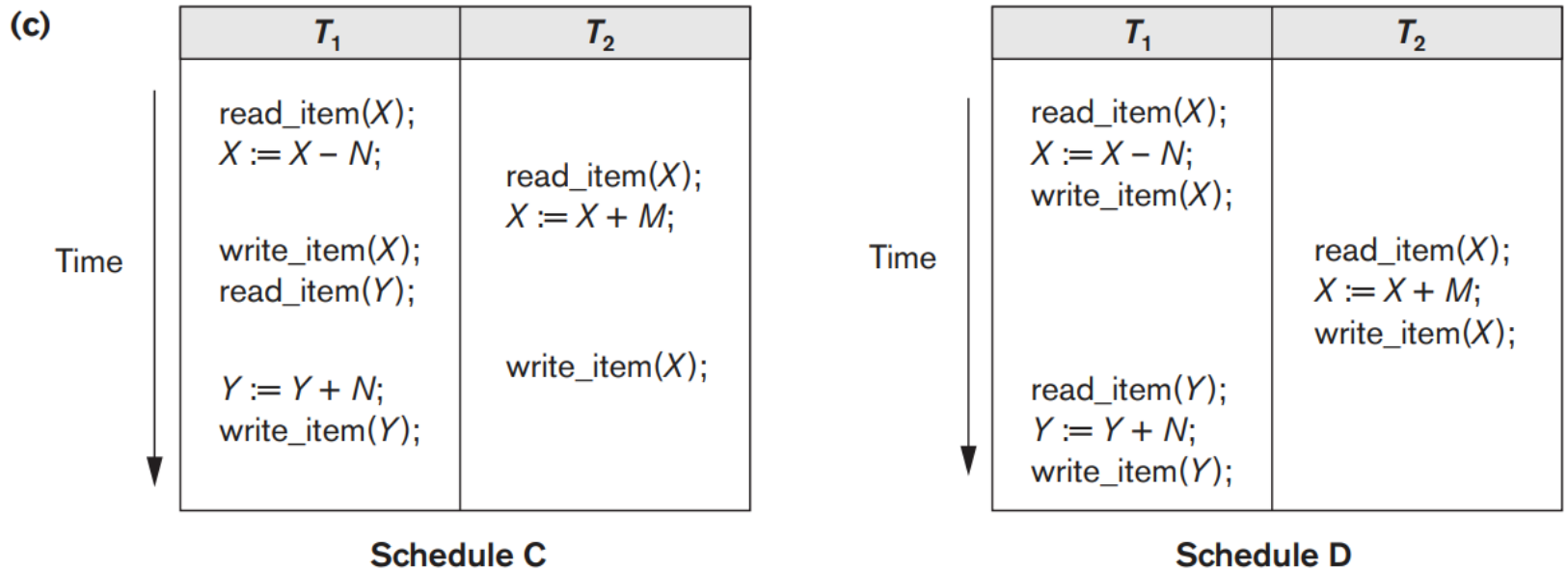
Serial Schedule

- If no interleaving of operations is permitted, there are only two possible outcomes:
  - Execute all the operations of transaction T1 (in sequence) followed by all the operations of transaction T2 (in sequence).
  - Execute all the operations of transaction T2 (in sequence) followed by all the operations of transaction T1 (in sequence).



Non Serial Schedule

- Formally, 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 **non-serial**.



Serialize Schedule

- A schedule S of n transactions is serializable if it is equivalent to some serial schedule of the same n transactions.
- Saying that a non-serial schedule S is serializable is equivalent to saying that it is correct, because it is equivalent to a serial schedule, which is considered correct. The remaining question is: When are two schedules considered equivalent?
- Two approaches:
  1. **result equivalent**
  2. For two schedules to be equivalent, the operations applied to each data item affected by the schedules should be applied to that item in both schedules in the same order.
- Two definitions of equivalence of schedules are generally used:
  - conflict equivalence
  - view equivalence.

Conflict Equivalence of Two Schedules.

- Two schedules are said to be conflict equivalent if the relative order of any two conflicting operations is the same in both schedules.
- If two conflicting operations are applied in different orders in two schedules, the effect can be different on the database or on the transactions in the schedule, and hence the schedules are not conflict equivalent.

Testing for Serializability of a Schedule

Testing Conflict Serializability of a Schedule S

1. For each transaction T_i participating in schedule S, create a node labeled T_i in the precedence graph.
2. For each case in S where T_j executes a read_item(X) after T_i executes a write_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
3. For each case in S where T_j executes a write_item(X) after T_i executes a read_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
4. For each case in S where T_j executes a write_item(X) after T_i executes a write_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
5. The schedule S is serializable if and only if the precedence graph has no cycles.

(a)

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

Time

Schedule A

(b)

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

Time

Schedule B

(c)

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);	

Time

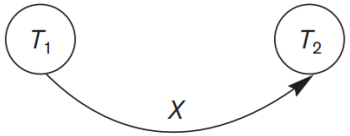
Schedule C

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);	

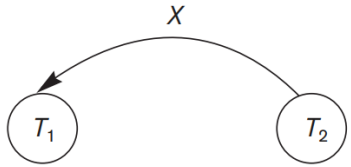
Time

Schedule D

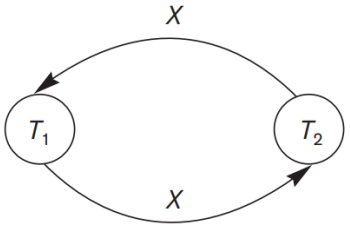
(a)



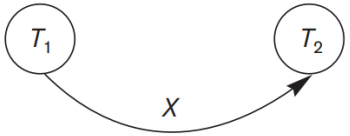
(b)



(c)



(d)



(a)

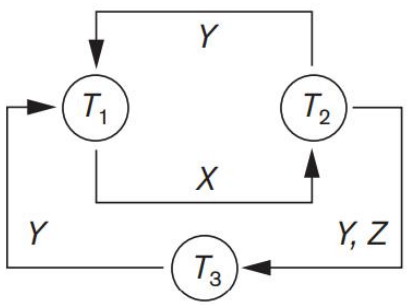
Transaction T_1	Transaction T_2	Transaction T_3
read_item(X);	read_item(Z);	read_item(Y);
write_item(X);	read_item(Y);	read_item(Z);
read_item(Y);	write_item(Y);	write_item(Y);
write_item(Y);	read_item(X);	write_item(Z);
	write_item(X);	

(b)

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

Schedule E

(d)



Equivalent serial schedules

None

Reason

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

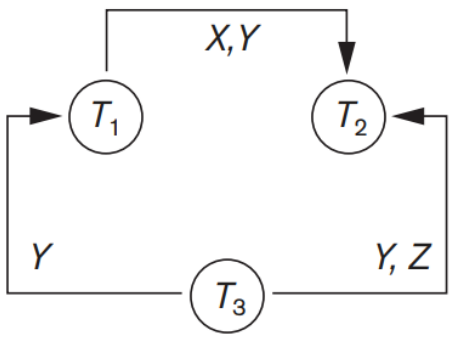
(c)

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

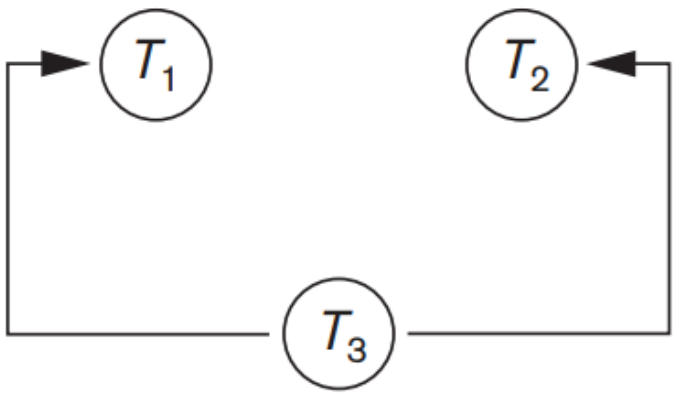
(e)



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

(f)



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

$T_3 \rightarrow T_2 \rightarrow T_1$

View serializability

- Two schedules S and S′ are said to be view equivalent if the following three conditions hold:

1. The same set of transactions participates in S and S′, and S and S′ include the same operations of those transactions.
2. For any operation $r_i(X)$ of T_i in S, if the value of X read by the operation has been written by an operation $w_j(X)$ of T_j (or if it is the original value of X before the schedule started), the same condition must hold for the value of X read by operation $r_i(X)$ of T_i in S′.
3. If the operation $w_k(Y)$ of T_k is the last operation to write item Y in S, then $w_k(Y)$ of T_k must also be the last operation to write item Y in S′.

Schedule S1:
$w_1(X);r_2(X);w_2(X);r_3(X);w_3(X);$

Schedule S2:
$w_1(X);r_2(X);\ w_2(X);r_3(X);w_3(X);$

- To check for view equivalence:

1. In S_1, T_2 reads X written by T_1
2. In S_2, T_2 also reads X written by T_1.

3. In S_1, T_3 performs the final write on X.
4. In S_2, T_3 also performs the final write on X.

5. In S_1, T_2 reads X written by T_1 , and T_3 writes the final value.
6. In S_2, the same sequence is maintained.

Since all conditions are satisfied, S_1 and S_2 are **view equivalent**.

Constrained Write Assumption

- The definitions of conflict serializability and view serializability are similar if a condition known as the **constrained write assumption (or no blind writes)** holds on all transactions in the schedule.
- The **constrained write assumption** states that:
  - Any **write operation** $w_i(X)$ in a transaction T_i must be **preceded by a read operation** $r_i(X)$ in the same transaction.
  - The value written by $w_i(X)$ is **dependent on the value read by $r_i(X)$.** Specifically, the new value written is computed as:

$$X_{new}=f(X_{old})$$

- A **blind write** is a write operation in a transaction *T* on an item *X* that is not dependent on the old value of *X*, so it is not preceded by a read of *X* in the transaction *T*.

View Equivalence and View Serializability

- Conflict serializability like view serializability if constrained write assumption (no blind writes) applies
- Less restrictive than conflict equivalence under the ability of unconstrained write assumption
- Unconstrained write assumption
  - Value written by an operation can be independent of its old value

$$S_g: r_1(X); w_2(X); w_1(X); w_3(X); c_1; c_2; c_3;$$

- Debit-credit transactions
  - Less-stringent conditions than conflict serializability or view serializability

Transaction Support in SQL

- No explicit Begin_Transaction statement
- Every transaction must have an explicit end statement
  - COMMIT
  - ROLLBACK
- Access mode is READ ONLY or READ WRITE
- Diagnostic area size option
  - Integer value indicating number of conditions held simultaneously in the diagnostic area

Characteristics of transaction in SQL

- Every transaction has certain characteristics attributed to it.
- The characteristics are the *access mode*, the *diagnostic area size*, and the *isolation level*.

Access Mode

- The **access mode** can be specified as READ ONLY or READ WRITE. The default is READ WRITE, unless the isolation level of READ UNCOMMITTED is specified in which case READ ONLY is assumed.
- A mode of READ WRITE allows select, update, insert, delete, and create commands to be executed.
- A mode of READ ONLY, as the name implies, is simply for data retrieval.

Diagnostic area size

- The **diagnostic area size** option, DIAGNOSTIC SIZE *n*, specifies an integer value *n*, which indicates the number of conditions that can be held simultaneously in the diagnostic area.
- These conditions supply feedback information (errors or exceptions)to the user or program on the *n* most recently executed SQL statement.

Isolation level

- The **isolation level** option is specified using the statement ISOLATION LEVEL <isolation>, where the value for <isolation> can be READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, or SERIALIZABLE

If a transaction executes at a lower isolation level than SERIALIZABLE, then one or more of the following three violations may occur:

Dirty read

- A transaction *T*1 may read the update of a transaction *T*2, which has not yet committed. If *T*2 fails and is aborted, then *T*1 would have read a value that does not exist and is incorrect.

Nonrepeatable read

- A transaction *T*1 may read a given value from a table.
- If another transaction *T*2 later updates that value and *T*1 reads that value again, *T*1 will see a different value.

Phantoms

- A transaction *T*1 may read a set of rows from a table, perhaps based on some condition specified in the SQL WHERE-clause.
- Now suppose that a transaction *T*2 inserts a new row *r* that also satisfies the WHERE-clause condition used in *T*1, into the table used by *T*1. The record *r* is called a **phantom record** because it was not there when *T*1 starts but is there when *T*1 ends. *T*1 may or may not see the phantom, a row that previously did not exist. If the equivalent serial order is *T*1 followed by *T*2, then the record *r* should not be seen; but if it is *T*2 followed by *T*1,then the phantom record should be in the result given to *T*1.
- If the system cannot ensure the correct behavior, then it does not deal with the phantom record problem.

Transaction Support in SQL (cont'd.)

	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

Possible violations based on isolation levels as defined in SQL

SQL Code Example

```
EXEC SQL WHENEVER SQLERROR GOTO UNDO;
EXEC SQL SET TRANSACTION
    READ WRITE
    DIAGNOSTIC SIZE 5
    ISOLATION LEVEL SERIALIZABLE;
EXEC SQL INSERT INTO EMPLOYEE (Fname, Lname, Ssn, Dno, Salary)
    VALUES ('Robert', 'Smith', '991004321', 2, 35000);
EXEC SQL UPDATE EMPLOYEE
    SET Salary = Salary * 1.1 WHERE Dno = 2;
EXEC SQL COMMIT;
GOTO THE_END;
UNDO: EXEC SQL ROLLBACK;
THE_END: ... ;
```

Transaction Support in SQL (cont'd.)

- Snapshot isolation
  - Used in some commercial DBMSs
  - Transaction sees data items that it reads based on the committed values of the items in the database snapshot when transaction starts
  - Ensures phantom record problem will not occur

Exercises

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

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

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

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

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