



# Chapter 4: Transaction Management

Adama Science and Technology University  
School of Electrical Engineering and Computing  
Department of CSE  
CSEg 2208: Database Systems  
(2022)

# Outline

- ❖ Introduction to Transaction Processing
- ❖ Transaction and System Concepts
- ❖ Desirable Properties of Transactions
- ❖ Characterizing Schedules based on Serializability
- ❖ Characterizing Schedules based on Recoverability
- ❖ Transaction Support in SQL

# Introduction to Transaction Processing

## ❖ Single-User System:

- ❖ At *most one user* at a time can use the database management system.
- ❖ Eg. Personal computer system.

## ❖ Multiuser System:

- ❖ *Many users* can access the *DBMS concurrently*.
- ❖ Eg. Air line reservation, Bank and the like system are operated by *many users* who submit transaction concurrently to the system.
- ❖ This is achieved by *multiprogramming*, which allows the OS of the computer to *execute multiple programs/processes* at the same time.

# Introduction to Transaction Processing

## ❖ Concurrency

### ❖ Interleaved processing:

- ❖ Concurrent execution of processes is interleaved in a *single CPU* using for example, *round robin algorithm*.

### ❖ Advantages:

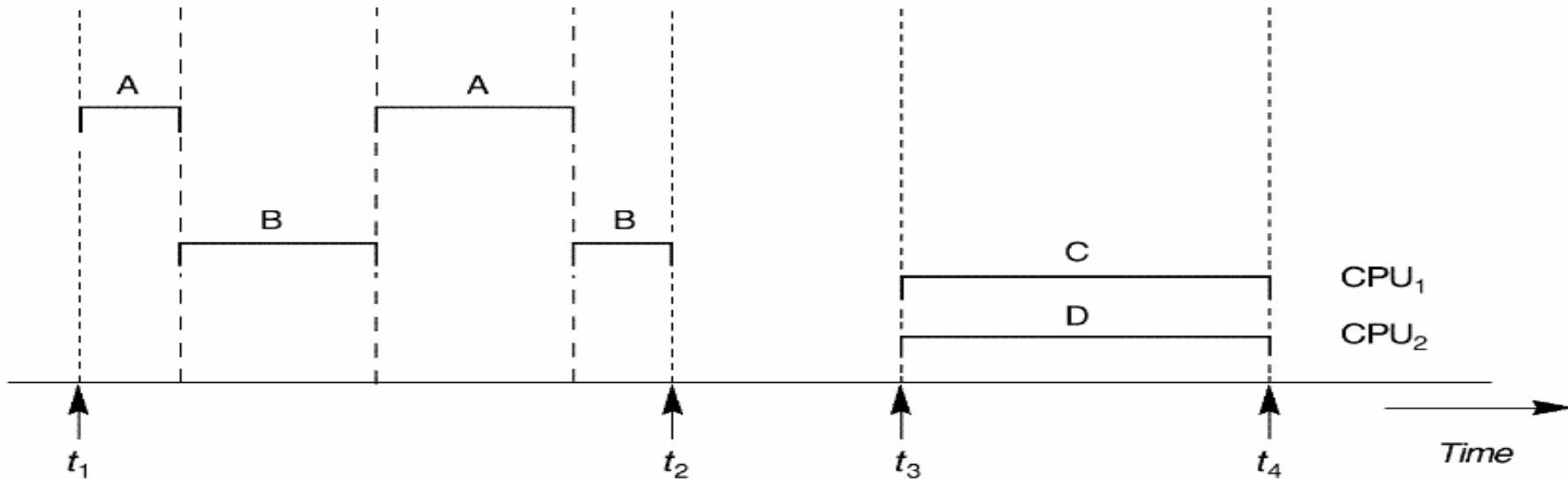
- ❖ Keeps the CPU busy when the process requires I/O by switching to *execute another process rather than remaining idle* during I/O time and hence this will increase system *throughput* (average no. of transactions completed within a given time)
- ❖ Prevents long process from delaying other processes (*minimize unpredictable delay in the response time*).



# Introduction to Transaction Processing

## Parallel processing:

- If Processes are concurrently executed in *multiple CPUs*.



Interleaved processing versus 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).
- ❖ Examples include *ATM transactions, credit card approvals, flight reservations, hotel check-in, phone calls, supermarket scanning, academic registration and billing.*

## ❖ Transaction boundaries

- ❖ Any *single transaction* in an application program is bounded with *Begin* and *End* statements.
- ❖ An *application program* may contain several transactions separated by the *Begin* and *End* transaction boundaries.

# 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 that measure the size of the data item.
- ❖ Basic operations that a transaction can perform 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. Basic unit of data transfer from the disk to the computer main memory is **one block**.

# Introduction to Transaction Processing

- ❖ **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.
- ❖ **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.



# Introduction to Transaction Processing

- ❖ *Store* the *updated* block from the buffer back to disk (either immediately or at some later point in time).
- ❖ The DBMS maintains a number of buffers in the *main memory* that holds *database disk blocks* which contains the database items being processed.
  - ❖ When this buffers are occupied and
  - ❖ If there is a need for additional database block to be copied to the main memory
- ❖ Some *buffer management policy* is used to choose for replacement but if the chosen buffer has been modified, it must be *written back to disk* before it is used.

# Introduction to Transaction Processing

## ❖ 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);
```

# Introduction to Transaction Processing

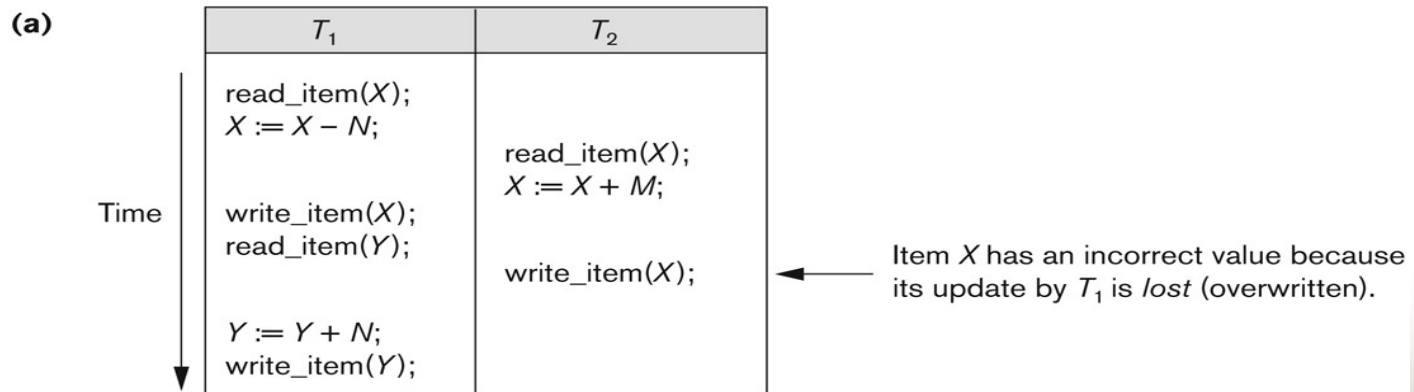
## ❖ Why Concurrency Control is needed: Three cases

### i. 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.

**Figure 1 .3**

Some problems that occur when concurrent execution is uncontrolled.



# Introduction to Transaction Processing

## i. The Lost Update Problem

E.g. Account with balance  $A=100$ .

- ❖ T1 reads the account A
- ❖ T1 withdraws 10 from A
- ❖ T1 makes the update in the Database
- ❖ T2 reads the account A
- ❖ T2 adds 100 on A
- ❖ T2 makes the update in the Database

- ❖ If the above case is done one after the other (serially), there is no problem.
  - ❖ If the execution is T1 followed by T2 then  $A=190$
  - ❖ If the execution is T2 followed by T1 then  $A=190$

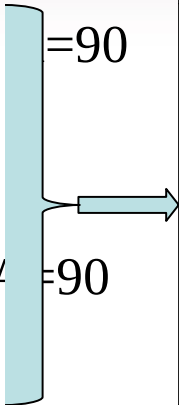


# Introduction to Transaction Processing

## i. The Lost Update Problem

❖ But if they start at the same time in the following sequence:

- ❖ T1 reads the account  $A=100$
- ❖ T1 withdraws 10 making the balance  $=90$
- ❖ T2 reads the account  $A=100$
- ❖ T2 adds 100 making  $A=200$
- ❖ T1 makes the update in the Database  $A=90$
- ❖ T2 makes the update in the Database  $A=200$



T1	T2
Read_item(A) $A=A-10$	
	Read_item(A) $A=A+100$
Write_item(A)	
	Write_item(A)

❖ After the successful completion of the operation the final value of  $A$  will be 200 which override the update made by the first transaction that changed the value from 100 to 90.

# Introduction to Transaction Processing

## ii. 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.  
Based on the above scenario:

Fig 2: Temporal update problem

T1	T2
Read_item(A) A=A-10 Write_item(A)  Commit	Read_item(A) A=A+100  Write_item(A)  Abort

Transaction T2 fails and must change the values of A back to its old value; Meanwhile T1 has read the temporary incorrect value of A

- ❖ T2 increases **A** making it 200 but then aborts the transaction before it is committed. T1 gets 200, subtracts 10 and make it 190. But the actual balance should be 90

# Introduction to Transaction Processing

## iii. *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.
- ❖ **Example:** T1 would like to add the values of A=10, B=20 and C=30. after the values are read by T1 and before its completion, T2 updates the value of B to be 50. at the end of the execution of the two transactions T1 will come up with the sum of 60 while it should be 90 since B is updated to 50

T1	T2
Sum= 0; Read_item(A) Sum=Sum+A Read_item(B) Sum=Sum+B  Read_item(C) Sum=Sum+C	       Read_item(B) B=50

# Introduction to Transaction Processing

## ❖ Why recovery is needed?

- ❖ Whenever a transaction is submitted to the DBMS for execution, the system is responsible for making sure that either
  - ❖ all operations in the transaction to be **completed successfully** or
  - ❖ the transaction has **no effect on the database or any other transaction.**
- ❖ The DBMS may permit some operations of a transaction T to be applied to the database but a transaction may **fail** after executing some of its operations



# Introduction to Transaction Processing

## What causes a Transaction to fail?

### ❖ 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.

### ❖ 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

- ❖ **Exception conditions detected by the transaction**
  - ❖ Certain conditions forces cancellation of the transaction.
  - ❖ For example, data for the transaction may not be found. such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal from that account, to be canceled.
- ❖ **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

## ❖ 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.

## ❖ Physical problems and catastrophes

- ❖ This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, overwriting disks or tapes by mistake

# 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** -indicates the beginning of a transaction execution
  - ❖ **Partially committed state** shows the end of read/write operation but this will not ensure permanent modification on the data base

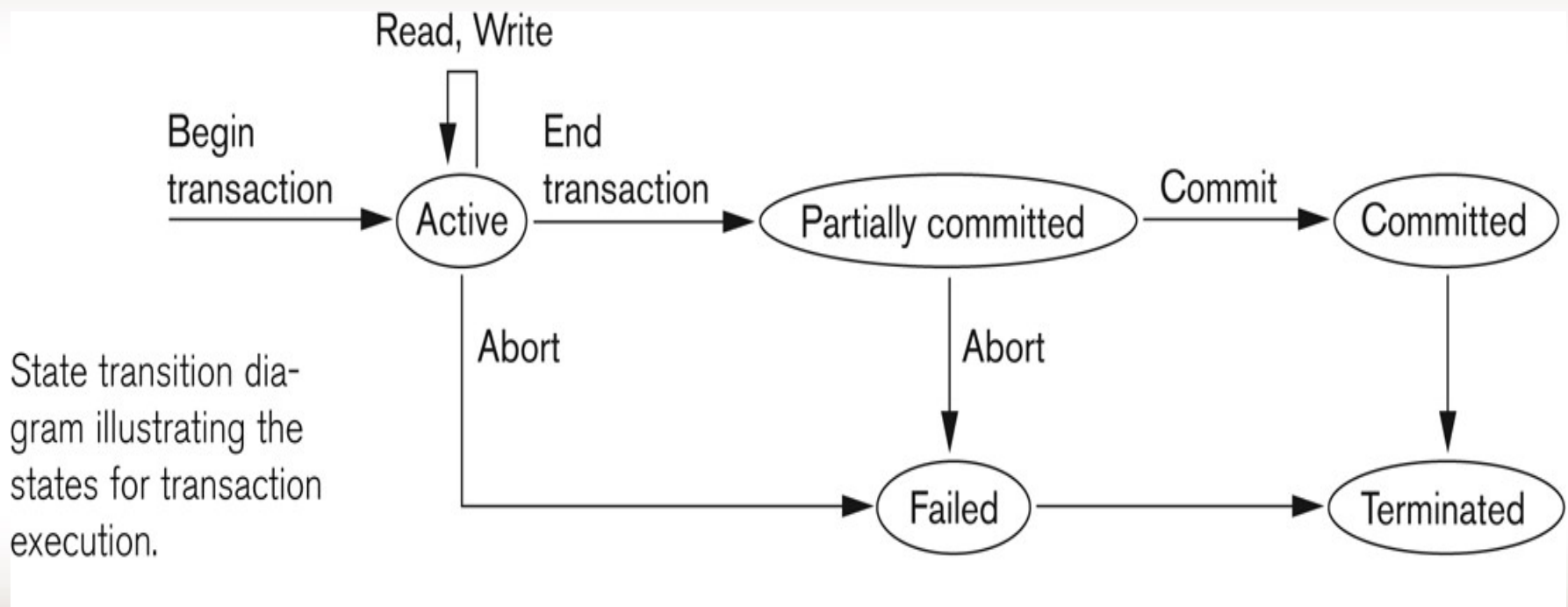


# Transaction and System Concepts

- ❖ **Committed state** -ensures that all the changes done on a record by a transaction were done persistently
- ❖ **Failed state** happens when a transaction is aborted during its active state or if one of the rechecking fails
- ❖ **Terminated State** -corresponds to the transaction leaving the system

# Transaction and System Concepts

## ❖ State transition diagram illustrating the states for transaction execution



# Transaction and System Concepts

- ❖ 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.

# Desirable Properties of Transactions

- ❖ To ensure data integrity , DBMS should maintain the following **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.



# Desirable Properties of Transactions

## Example:

❖ Suppose that **T<sub>i</sub>** is a transaction that transfer 200 birr from account CA2090( which is 5,000 Birr) to SB2359(which is 3,500 birr) as follows

- ❖ **Read(CA2090)**
- ❖ **CA2090= CA2090-200**
- ❖ **Write(CA2090)**
- ❖ **Read(SB2359)**
- ❖ **SB2359= SB2359+200**
- ❖ **Write(SB2359)**

❖ **Atomicity**- either all or none of the above operation will be done – this is materialized by **transaction management** component of DBMS

# Desirable Properties of Transactions

- ❖ **Consistency**-the sum of CA2090 and SB2359 be unchanged by the execution of  $T_i$  i.e 8500- this is the responsibility of **application programmer** who codes the transaction.
- ❖ **Isolation**- when several transaction are being processed concurrently on a data item they may create many inconsistent problems. So handling such case is the responsibility of **Concurrency control** component of the DBMS.
- ❖ **Durability** - once  $T_i$  writes its update this will remain there when the database restarted from failure . This is the responsibility **of recovery management** components of the DBMS.

# Characterizing Schedules

## ❖ 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 $T_1, T_2, \dots, T_n$ :

❖ It is an ordering of the operations of the transactions subject to the constraint that, for each transaction  $T_i$  that participates in  $S$ , the operations of  $T_i$  in  $S$  must appear in the same order in which they occur in  $T_i$ .

❖ Note, however, that operations from other transactions  $T_j$  can be interleaved with the operations of  $T_i$  in  $S$ . **Eg. Consider the following example:**

$S_a$ :  $r_2(A); w_2(A); r_1(A); w_1(A); a_2$ ;

❖ **Schedule:** A sequence of the operations by a set of concurrent transactions that preserves the order of the operations in each of the individual transactions.

T1	T2
Read_item(A) A=A-10 Write_item(A)	Read_item(A) A=A+100 Write_item(A)  Abort

# Characterizing Schedules

❖ Two operations in a schedule are said to be in **conflict** if they satisfy one of the following conditions.

- ❖ They belong to different transactions
- ❖ They access the same data item X
- ❖ At least one of the operations is a write\_Item(X)

E.g., Sa: r1(X); r2(x); w1(X); r1(Y); W2(X); W1(Y);

- ❖ r1(X) and w2(X)
  - ❖ r2(X) and w1(X);
  - ❖ W1(X) and w2(X)
- } **Conflicting operations**

- ❖ r1(X) and r2(X)
  - ❖ W2(X) and w1(Y)
  - ❖ r1(x) and w1(x)
- } **No Conflicting, why?**

## **Non-serial schedule:**

- A schedule where the operations from a set of concurrent *transactions* *schedule* are *interleaved*.

## **Serial Schedule:**

- A schedule where the operations of each transaction are *executed consecutively* without any *interleaved operations* from other transactions.



# Characterizing Schedules based on recoverability:

## ❖ Recoverable schedule:

- ❖ One where no *committed 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.

## ❖ Examples:

❖ Sc: r1(X); w1(X); r2(X); r1(Y); w2(x); c2; a1; **not recoverable**

❖ Sd: r1(X); w1(X); r2(X); r1(Y); w2(X); w1(Y); c1; c2;

❖ Se: r1(X); w1(X); r2(X); r1(Y); w2(x) ; w1(Y); a1; a2;

**Recoverable**

Where:

a → abort,

c → commit

# Characterizing Schedules based on Recoverability:

## ❖ Cascadeless schedule:

❖ One where every transaction *reads* only the items that are *written* by *committed* transactions.

❖ Examples

❖ **Sf:**  $r_1(X); w_1(X); r_1(Y); c_1; r_2(X); w_2(X); w_1(Y); c_2;$

## ❖ 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/aborted*.

❖ Examples:

**Sg:**  $w_1(X,5); c_1; w_2(x,8);$

# Characterizing Schedules based on Recoverability

- ❖ The concept of *Serializable of schedule* is used to identify **which schedules are correct** when concurrent transactions executions have interleaving of their operations in the schedule.
- ❖ **Serial schedule:**
  - ❖ A schedule  $S$  is **serial** if, for every transaction  $T$  participating in the schedule, all the operations of  $T$  are *executed consecutively* in the schedule.
  - ❖ Otherwise, the schedule is called *nonserial schedule*.
  - ❖ *For example*, in the banking example suppose there are two transaction where one transaction *calculate the interest* on the account and another *deposit some money* into the account.

# Characterizing Schedules based on Recoverability

❖ Hence the *order of execution* is important for the *final result*.

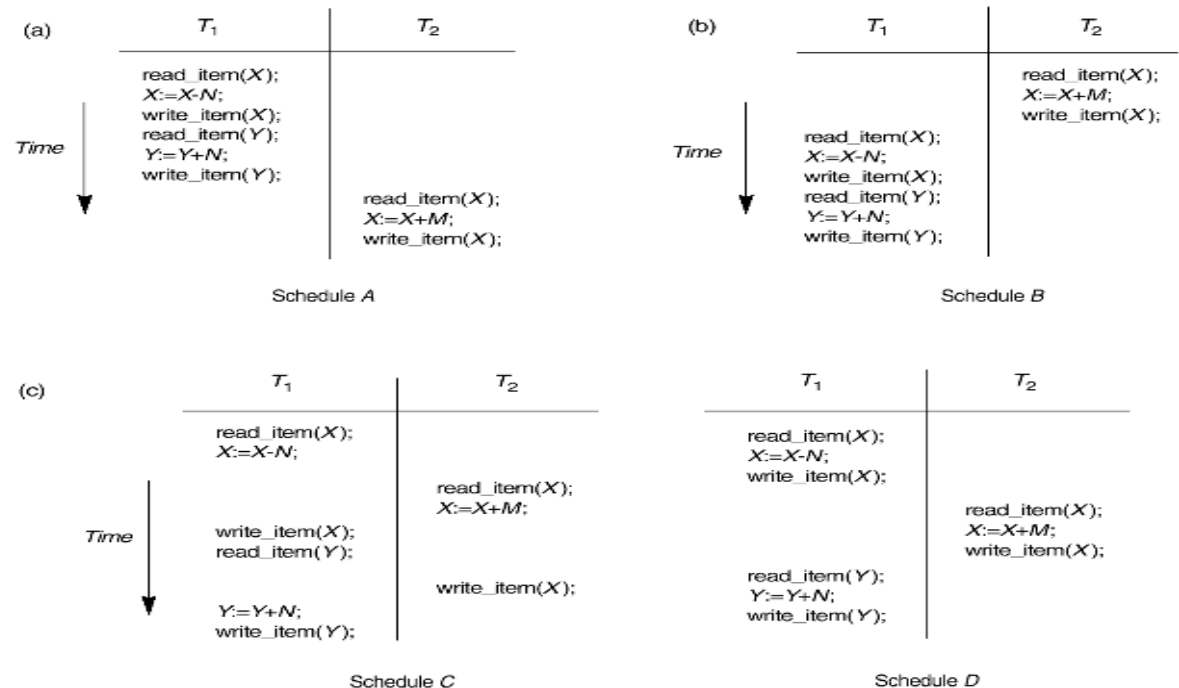
## ❖ Serializable schedule

- ❖ A schedule whose effect on any consistent database instance is identical to that of some complete **serial schedule** over the set of *committed* transactions in  $S$ .
- ❖ A nonserial schedule  $S$  is serializable, is equivalent to say that it is **correct** with the result of one of the serial schedule.



# Characterizing Schedules based on Recoverability

## ❖ Example



Examples of serial and nonserial schedules involving transactions  $T_1$  and  $T_2$ .

# Characterizing Schedules based on Recoverability

## ❖ Result equivalent

- ❖ Two schedules are called result equivalent if they produce the same final state of the database
- ❖ Two types of equivalent schedule: **Conflict and view**

## ❖ Conflict equivalent

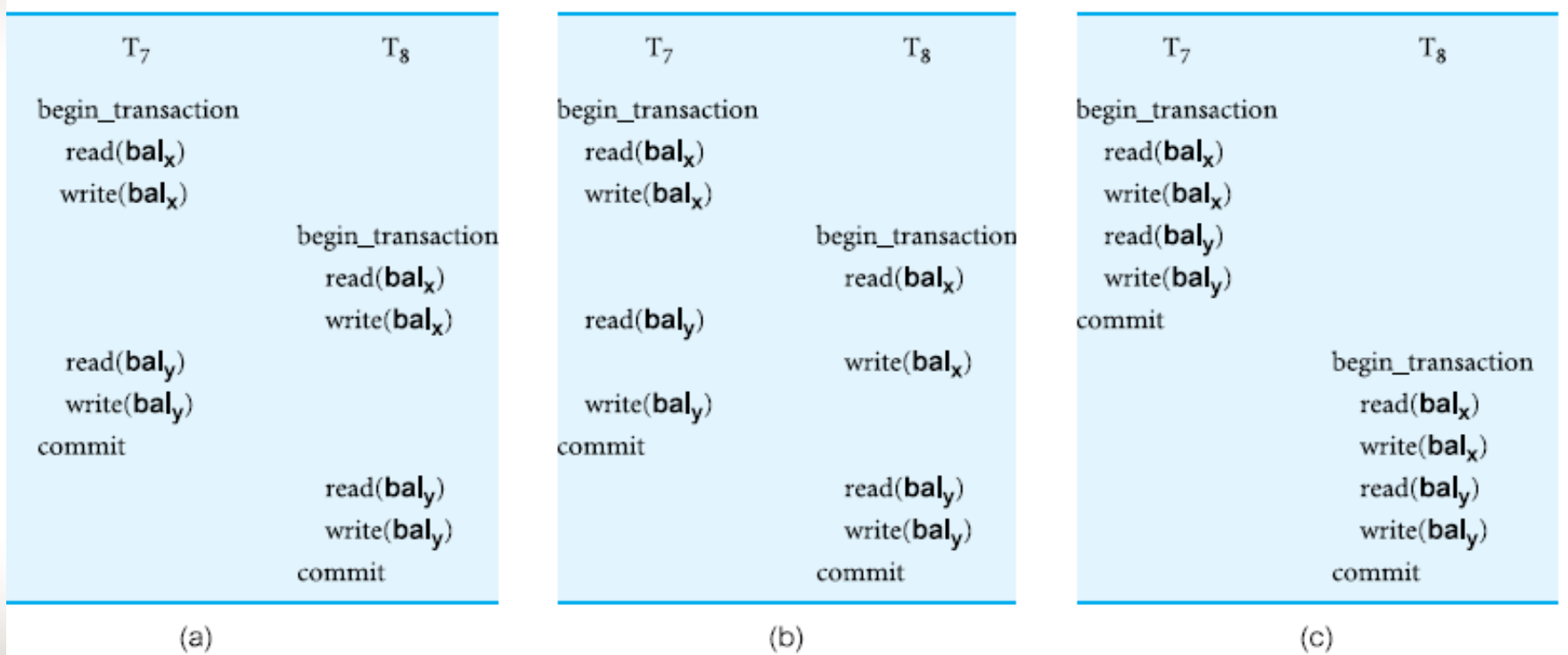
- ❖ Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both schedules.

## ❖ Examples

- ❖ S1: r1(x); w2(x) & S2: w2(x); r1(x)
  - ❖ S1: w1(x); w2(x); & S2: w2(x); w1(x);
- } **Not conflict equivalent**

# Characterizing Schedules based on Recoverability

Equivalent schedules: (a) nonserial schedule S1; (b) nonserial schedule S2 equivalent to S1; (c) serial schedule S3, equivalent to S1 and S2.



# Characterizing Schedules based on Recoverability

## ❖ Conflict Serializable:

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



# Characterizing Schedules based on Recoverability

- If you can transform an interleaved schedule by swapping consecutive non-conflicting operations of different transactions into a serial schedule, then the original schedule is **conflict serializable**.
- *Example:*

$$\begin{array}{ccc} s_2: R(A) & W(A) & R(B) & W(B) \\ & R(A) & W(A) & R(B) & W(B) \\ & & \equiv & \\ s_1: R(A) & W(A) & R(B) & W(B) \\ & & R(A) & W(A) & R(B) & W(B) \end{array}$$

# Characterizing Schedules based on Recoverability

ii. Two schedules are said to be view equivalent if the following three conditions hold:

- ❖ The same set of transactions participates in S and S', and S and S' include the same operations of those transactions.
- ❖ If  $T_i$  reads a value  $A$  written by  $T_j$  in S1, it must also read the value of  $A$  written by  $T_j$  in S2
- ❖ for each data object  $A$ , the transaction that perform the final write on  $A$  in S1 must also perform the final write on  $A$  in S2

S'

T1:	R(A)	W(A)	
T2:		W(A)	
T3:			W(A)

view  
≡

S

T1:	R(A),W(A)		
T2:		W(A)	
T3:			W(A)

# Characterizing Schedules based on Recoverability

## Testing for conflict serializable Algorithm

- ❖ Looks at only **read\_Item (X)** & **write\_Item (X)** operations
- ❖ Constructs a precedence graph (**serialization graph**) - a graph with directed edges
- ❖ An edge is created from **T<sub>i</sub>** to **T<sub>j</sub>** if one of the operations in T<sub>i</sub> appears before a conflicting operation in T<sub>j</sub>
- ❖ The schedule is serializable if and only if the precedence graph has no cycles.

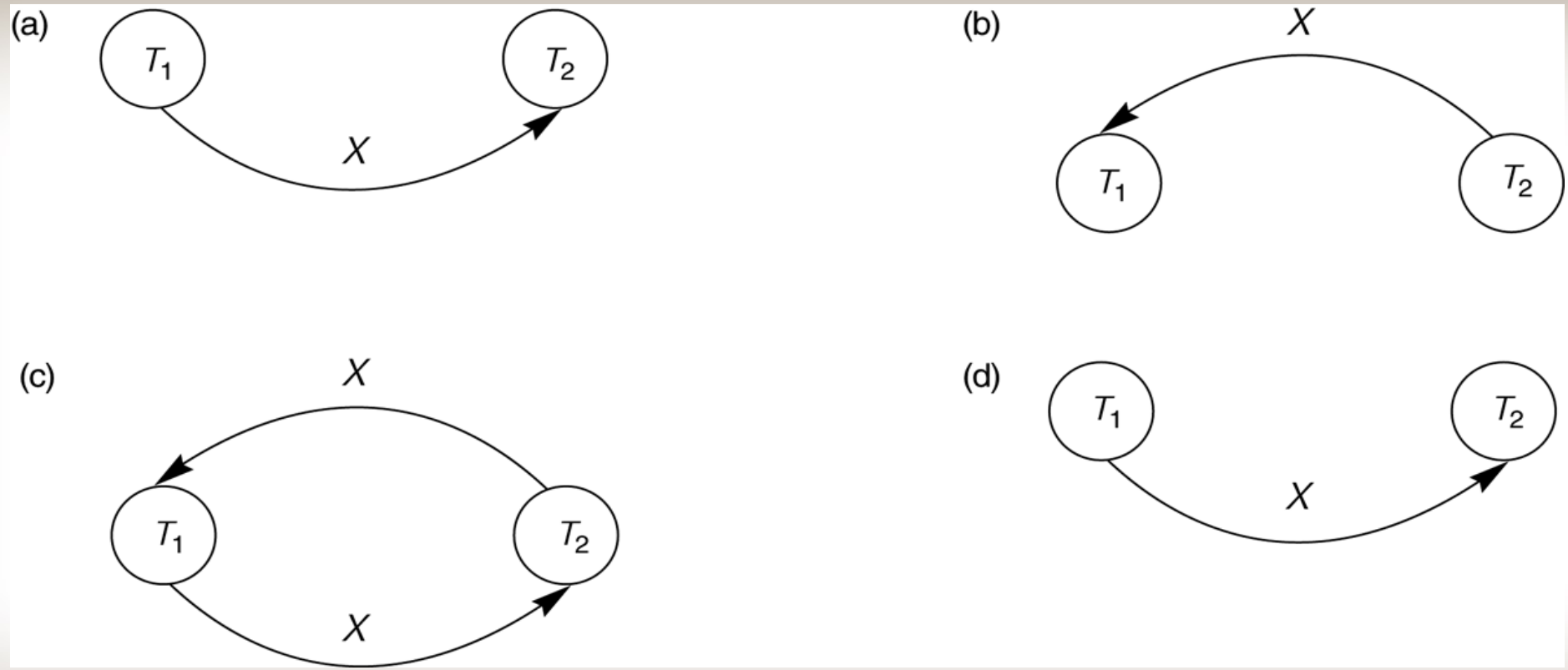
# Characterizing Schedules based on Recoverability

## Constructing the Precedence Graphs

- ❖ **Example:** Constructing the precedence graphs for schedules A to D from the previous Figure (Slide No. 33) to test for conflict serializability.
  - ❖ (a) Precedence graph for serial schedule A.
  - ❖ (b) Precedence graph for serial schedule B.
  - ❖ (c) Precedence graph for schedule C (not serializable).
  - ❖ (d) Precedence graph for schedule D (serializable, equivalent to schedule A).



# Characterizing Schedules based on Recoverability



# Characterizing Schedules based on Recoverability

## Another example of serializability Testing

(a)

Another example of serializability testing.  
(a) The read and write operations of three transactions  $T_1$ ,  $T_2$ , and  $T_3$ . (b) Schedule E. (c) Schedule F.

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);
```

# Characterizing Schedules based on Recoverability

Another example of serializability testing. (a) The read and write operations of three transactions  $T_1$ ,  $T_2$ , and  $T_3$ . (b) Schedule E. (c) Schedule F.

(b)

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

**Schedule E**

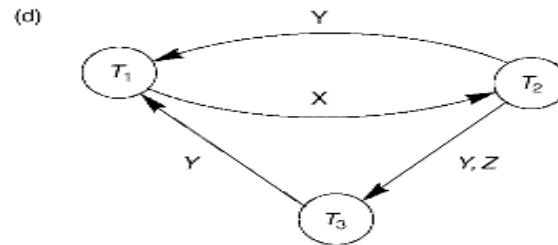
(c)

Another example of serializability testing. (a) The read and write operations of three transactions  $T_1$ ,  $T_2$ , and  $T_3$ . (b) Schedule E. (c) Schedule F.

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

**Schedule F**

# Characterizing Schedules based on Recoverability

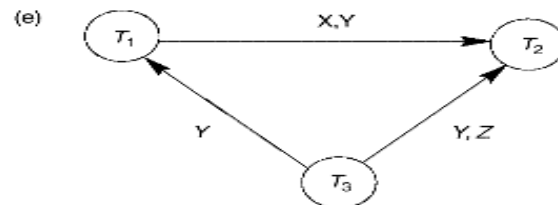


Equivalent serial schedules

None

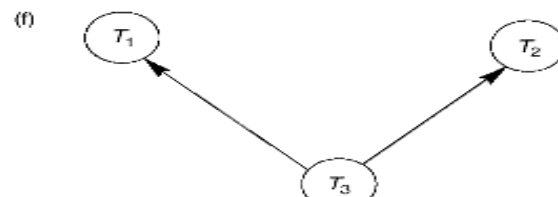
Reason

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



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$



Equivalent serial schedules

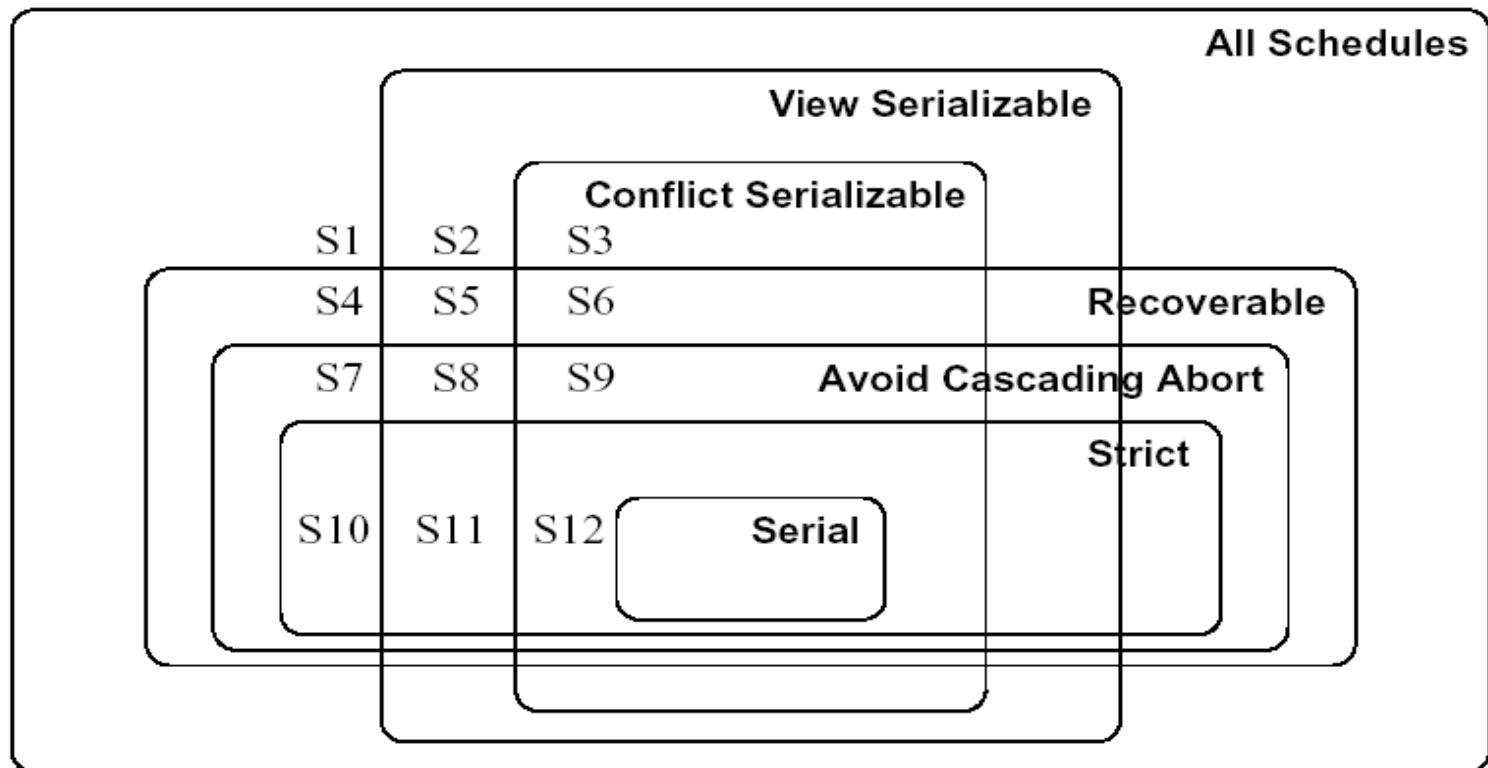
$T_3 \rightarrow T_1 \rightarrow T_2$

$T_3 \rightarrow T_2 \rightarrow T_1$

Another example of serializability testing. (d) Precedence graph for schedule E. (e) Precedence graph for schedule F. (f) Precedence graph with two equivalent serial schedules.



# Summery of Schedule types



Venn Diagram for Classes of Schedules



# Transaction Support in SQL

- ❖ A **single** SQL statement is always considered to be **atomic**.
  - ❖ Either the statement completes execution without error or it fails and leaves the database unchanged.
- ❖ Every transaction has three characteristics: **Access mode, Diagnostic size and isolation**
  - i. **Access mode:**
    - **READ ONLY or READ WRITE**
      - If the access mode is Read ONLY , INSERT, DELET , UPDATE & CREATE commands cannot be executed on the data base

# Transaction Support in SQL

- The default is READ WRITE unless the isolation level of READ UNCOMMITTED is specified, in which case READ ONLY is assumed.
- i. **Diagnostic size n**, specifies an integer value n, indicating the number of error conditions that can be held simultaneously in the diagnostic area.
- ii. **Isolation level** can be
  - » READ UNCOMMITTED,
  - » READ COMMITTED,
  - » REPEATABLE READ or
  - » SERIALIZABLE. The default is SERIALIZABLE.

# Transaction Support in SQL

- **Sample SQL transaction:**

```
EXEC SQL whenever sqlerror go to UNDO;  
EXEC SQL SET TRANSACTION  
    READ WRITE  
    DIAGNOSTICS 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

- With SERIALIZABLE: the interleaved execution of transactions will adhere to the notion of serializability.
- However, if any transaction executes at a lower level, then serializability may be violated.

# Transaction Support in SQL

## Potential problem with lower isolation levels: Four types

### i. Non repeatable Reads/unrepeatable Read: RW Conflicts

- A transaction  $T_2$  could change the value of an object  $A$  that has been read by a transaction  $T_1$ , while  $T_1$  is still in progress.
- If  $T_1$  tries to read the value  $a$  again it will get a different value

T1:	R(A),	R(A), W(A), C
T2:	R(A), W(A), C	

### ii. Reading Uncommitted Data ( “dirty reads”): WR Conflicts

- A transaction  $T_2$  could read a database object  $A$  that has been modified by another transaction  $T_1$ , which has not yet committed.

T1:	R(A), W(A),	R(B), W(B), Abort
T2:	R(A), W(A), C	



# Transaction Support in SQL

## iii. Overwriting Uncommitted Data: WW Conflicts

- ❖ A transaction  $T2$  could overwrite the value of an object  $A$ , which has already been modified by a transaction  $T1$ , while  $T1$  is still in progress.

T1:	W(A),	W(B), C
T2:	W(A), W(B), C	

## iv. Phantoms:

- ❖ New rows being read using the same read with a condition.
  - ❖ A transaction  $T1$  may read a set of rows from a table, perhaps based on some condition specified in the SQL WHERE clause.

# Transaction Support in SQL

- ❖ Now suppose that a transaction T2 inserts a new row that also satisfies the WHERE clause condition of T1, into the table used by T1.
- ❖ If T1 is repeated, then T1 will see a row that previously did not exist, called a **phantom**. Consider the following example code:

## **Transaction A begins.**

- `SELECT * FROM employee WHERE salary > 30000`

## **Transaction B begins.**

- `INSERT INTO employee (empno, firstname, midinit, lastname, job, salary) VALUES ('000350', 'NICK', 'A','GREEN','LEGAL COUNSEL',35000)`

**Transaction B inserts a row that would satisfy the query in Transaction A if it were issued again.**

# Transaction Support in SQL

- Possible violation of serializability

Read Phenomena	Dirty Reads	Lost Updates (Inconsistent)	Non-Repeatable Reads	Phantom
Isolation Level				
Read Uncommitted	yes	yes	yes	yes
Read Committed	No	Yes	yes	yes
Repeatable Read	No	No	No	Yes
Serializable	No	No	No	No

# Exercise One

1. Give an example schedule ,with lower isolation level , with actions of transactions  $T1$  and  $T2$  on objects  $X$  and  $Y$  that results in a **write-read** conflict.
2. Give an example schedule ,with lower isolation level , with with actions of transactions  $T1$  and  $T2$  on objects  $X$  and  $Y$  that results in a **read-write** conflict.
3. Give an example schedule ,with lower isolation level , with with actions of transactions  $T1$  and  $T2$  on objects  $X$  and  $Y$  that results in a **write-write** conflict.

# Exercise Two

- Consider the following transaction schedule .
- Is it Conflict Serializability?

T1	T2	T3
	R(X)	
		R(X)
W(Y)		
	W(X)	
		R(Y)
	W(Y)	

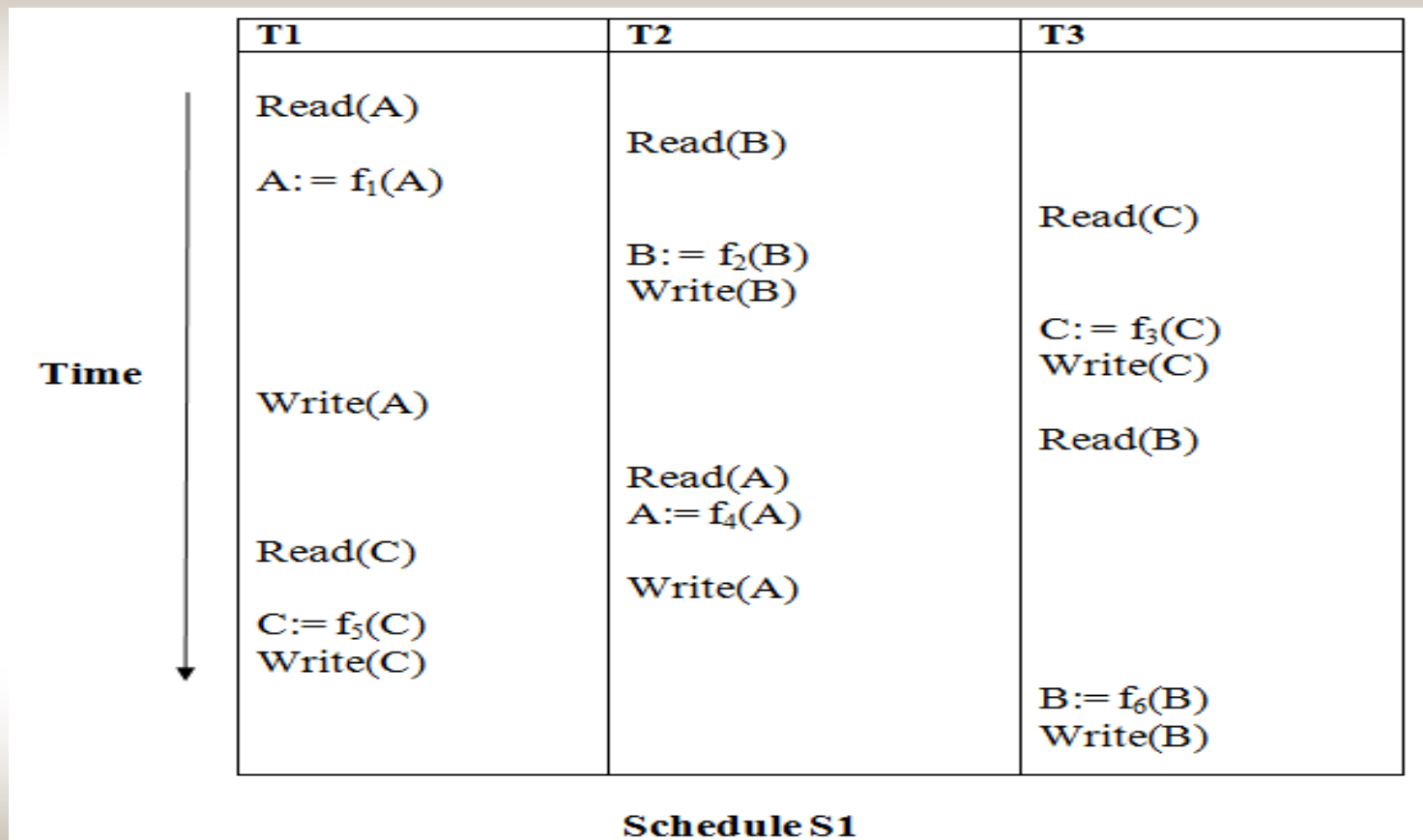
- Consider the following transaction schedule and test it for View Serializability.

T1	T2	T3
R(A)		
	W(A)	
		R(A)
W(A)		
		W(A)



# Exercise Three

- Determine whether the following schedule is serializable



# Question & Answer



**Thanks !!!**