Database Management System (15ECSC208)

UNIT II: Chapter 5: Transactions **(Part 1)**

Transaction Processing

Motivated by two independent reqs.

- **Concurrent DB access**
- **Resilience to system failures**

Concurrent Database Access

- **Attribute-level Inconsistency**

Client1: **Update** Project **Set** amount = amount + 10000 **where** projectID = 'DM22';

concurrent with …

Client2: **Update** Project **Set** amount = amount + 20000 **where** projectID = 'DM22';

Concurrent Database Access cont

- **Tuple-level Inconsistency**

Client1: **Update** Department **Set** phoneNo = '2378410' **where** deptID = '3'

concurrent with …

Client2: **Update** Department **Set** location = 'Main Building' **where** deptID = '3'

Concurrent Database Access cont

- **Table-level Inconsistency**

Client1: **Update** Professor **Set** decision = 'accept' **where** profID **in** (**select** profID **from** Project **where** amount > 20000;)

concurrent with …

Client2: **Update** Project **Set** amount = 1.1 * amount **where** startDate > 01/01/2010

- **Concurrency Goal**
- Execute sequence of SQL statements so that they appear to be running in isolation.

- **System-Failure Goal**
- Guarantee all-or-nothing execution, regardless of failures.

- **Solution for both concurrency and failures is TRANSACTION.**

Transactions

- **A transaction is a sequence of one or more SQL operations treated as a unit**
  - Transactions appear to run in isolation (Concurrency goal)
  - If the system fails, each transaction's changes are reflected either entirely or not at all (System-failure goal)
- **In terms of SQL standard:**
  - Transaction begins automatically on first SQL statement
  - On "**`commit`**" transaction ends and new one begins
  - Current transaction ends on session termination
  - "**`Autocommit`**" turns each statement into transaction

Types of Failures

- Computer Failure
  - A hardware or software error in the computer during transaction execution.

- Transaction Error
  - Failure caused by an operation in the transaction

Types of Failures cont…

- Local Errors
  - Conditions that cause the cancellation of transaction. (eg: data needed for transaction not found.)

- Concurrency control enforcement
  - Transaction may be aborted by the concurrency control method.

Types of Failures cont…

- Disk Failure
  - Loss of data in disk blocks during a transaction due to, disk read/write head crash.

- Physical problems and catastrophes
  - Problems like power failure, fire etc.

What is a Transaction?

- A transaction is an atomic unit of database access, which is either completely executed or not executed at all.

- "All or nothing" principle
  - Example: Online booking of railway reservation tickets

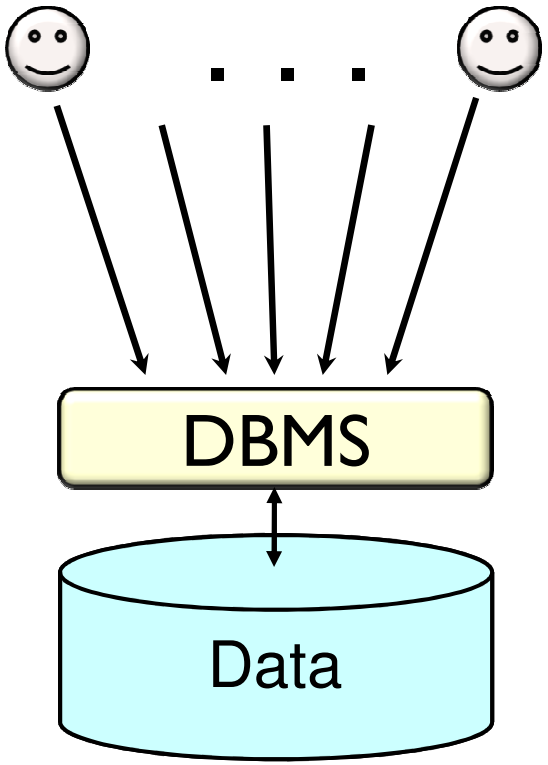
Transactions

- Many enterprises use databases to store information about their state
  - Eg: Balances of all depositors at a bank
- When an event occurs in the real world that changes the state of the enterprise, a program is executed to change the database state in a corresponding way
  - Eg: Bank balance must be updated when deposit is made
- A transaction is a program that accesses the database in response to real-world events.

Transaction Properties (ACID)

- **A**tomicity
- **C**onsistency
- **I**solation
- **D**urability

(ACID Properties) **Isolation**



Serializability

Operations may be interleaved, but execution must be equivalent to *some* sequential (serial) order of all transactions

How does the system guarantee Serializability

LOCKing

Concurrent Database Access

- **Attribute-level Inconsistency**

  T1: Update Project **Set** amount = amount + 10000 **where** projectID = 'DM22';

  concurrent with …

  T2: Update Project **Set** amount = amount + 20000 **where** projectID = 'DM22';

Concurrent Database Access cont

- **Tuple-level Inconsistency**

  T1: **Update** Department **Set** phoneNo = '2378410' **where** deptID = '3'

  concurrent with …

  T2: **Update** Department **Set** location = 'Main Building' **where** deptID = '3'

Concurrent Database Access cont

- **Table-level Inconsistency**

T1: **Update** Professor **Set** decision = 'accept' **where** profID **in** (**select** profID **from** Project **where** amount > 20000;)

concurrent with …

T2: **Update** Project **Set** amount = 1.1 * amount **where** startDate > 01/01/2010

Example of Transactions

- **Debit Transaction**
  - Debit(Account_Number, Withdraw_Amt)
  - *Begin_Transaction*
  - Read(Account_Number,Balance) (ID1)
  - Balance = Balance – Withdraw_Amt (ID2)
  - Write(Account_Number, Balance) (ID3)
  - *End_Transaction*

Example of Transactions cont…

- **Credit Transaction**
  - Credit(Account_Number, Credit_Amt)
  - *Begin_Transaction*
  - Read(Account_Number, Balance) (IC1)
  - Balance = Balance + Credit_Amt (IC2)
  - Write(Account_Number, Balance) (IC3)
  - *End_Transaction*

Scenario for Isolation

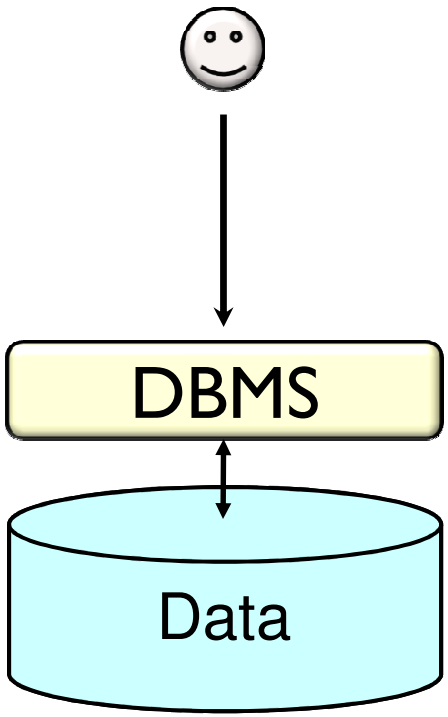
- What happens if the results of credit transaction are visible to debit transaction before it is actually written onto the database?

  Eg: IC2: writes Balance : read by ID1

- Transaction Properties (**Isolation**): the results of one transaction will not be visible to other transactions till the transaction "Commits".

- **Explanation**: Either debit or credit transaction results will not be available until they are committed.

(ACID Properties) **Durability**



If system crashes after transaction commits, all effects of transaction remain in database

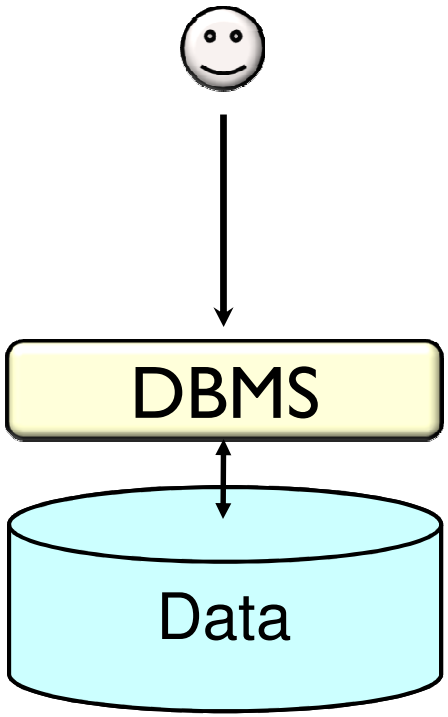
How does the system guarantee Durability

Logging

Scenario for Durability

- What happens if the database server crashes before the changed data is written onto a stable storage?
- Transaction Properties (**Durability**): The effects of committed transactions are not lost after commitment.
- **Explanation**: The effects are debit transaction will not be lost once it is committed.

(ACID Properties) **Atomicity**



Each transaction is "all-or-nothing," never left half done

How does the system guarantee Atomicity

Logging

(ACID Properties) **Atomicity**

- **Transaction Rollback (= Abort)**
  - Undoes partial effects of transaction
  - Can be system- or client-initiated

```
Begin Transaction;
<get input from user>
SQL commands based on input
<confirm results with user>
If ans='ok' Then Commit; Else Rollback;
```

Transaction work on Locking mechanisms

Scenario for Atomicity

- What happens if the credit transaction fails after executing IC1 and before executing IC2?

- Transaction Properties (**Atomicity**): Either all the instructions are executed in full or none.

- **Explanation**: In the case of credit transaction, all the instructions starting from begin transaction to end transaction (IC1, IC2, IC3) will be executed in full or none: Atomicity property.

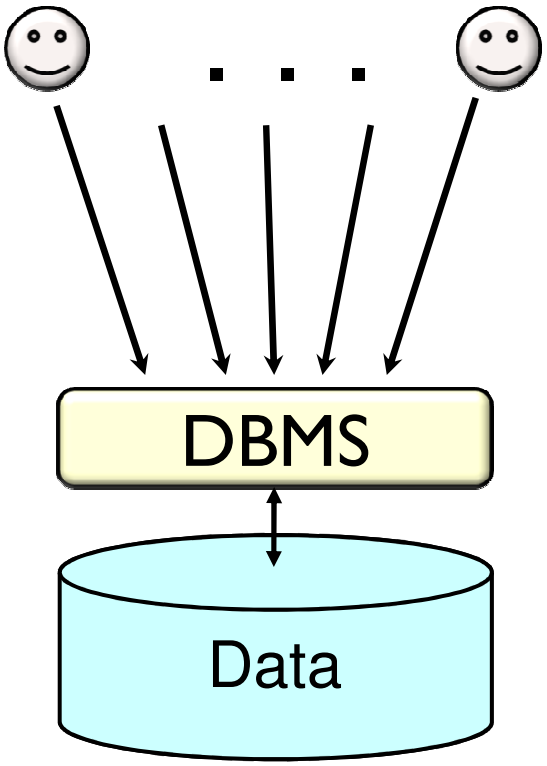
Example of Transactions

- **Credit Transaction**
  - Credit(Account_Number, Credit_Amt)
  - *Begin_Transaction*
  - Read(Account_Number, Balance) (IC1)
  - Balance = Balance + Credit_Amt (IC2)
  - Write(Account_Number, Balance) (IC3)
  - *End_Transaction*

Example of Transactions cont…

- **Debit Transaction**
  - Debit(Account_Number, Withdraw_Amt)
  - *Begin_Transaction*
  - Read(Account_Number,Balance) (ID1)
  - Balance = Balance – Withdraw_Amt (ID2)
  - Write(Account_Number, Balance) (ID3)
  - *End_Transaction*

(ACID Properties) **Consistency**



Each client, each transaction:

- Can assume all constraints hold

when transaction begins

- Must guarantee all constraints hold

when transaction ends

Serializability $\Rightarrow$ constraints always hold

Scenario for Consistency

- What happens if Credit transaction and Debit transaction execute simultaneously?
  - Eg: IC1, ID1, IC2, ID2, IC3, ID3

- Transaction Properties (**Consistency**): When multiple transactions are accessing data simultaneously, the access is protected through concurrency control mechanisms.

- **Explanation**: In case both debit transaction and credit transaction access the account number and balance data simultaneously, they will be protected through concurrency control mechanisms.

Read Operation – Read(X)

- Find the address of the disk block that contains item X.
- Copy that disk block into a buffer in main memory.
- Copy item X from the buffer to the program variable named X.

Write Operation – Write(X)

- Find the address of the disk block that contains X.
- Copy that disk block into a buffer in main memory.
- Copy item X from the program variable named X into the correct location in the buffer.
- Store the updated block from the buffer back to the disk.

Transaction Terminology

- **Commit** – Transaction completely performs all the read's and write's and changes the database state accordingly.

- **Abort** – Transaction is unable to complete all the read's and write's and hence will undo the operations that were performed till that point. Database will remain in the same state as it was prior to the beginning of this transaction.

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);
```

Why Concurrency Control is needed

- Multiple transactions are allowed to run **concurrently**.

- Advantages are:
  - **increased processor and disk utilization**, leading to better transaction throughput
    - one transaction can be using the CPU while another is reading from or writing to the disk
    - throughput: # of transactions executed in a given amount of time.
  - **reduced average response time** for transactions
    - short transactions need not wait behind long ones.

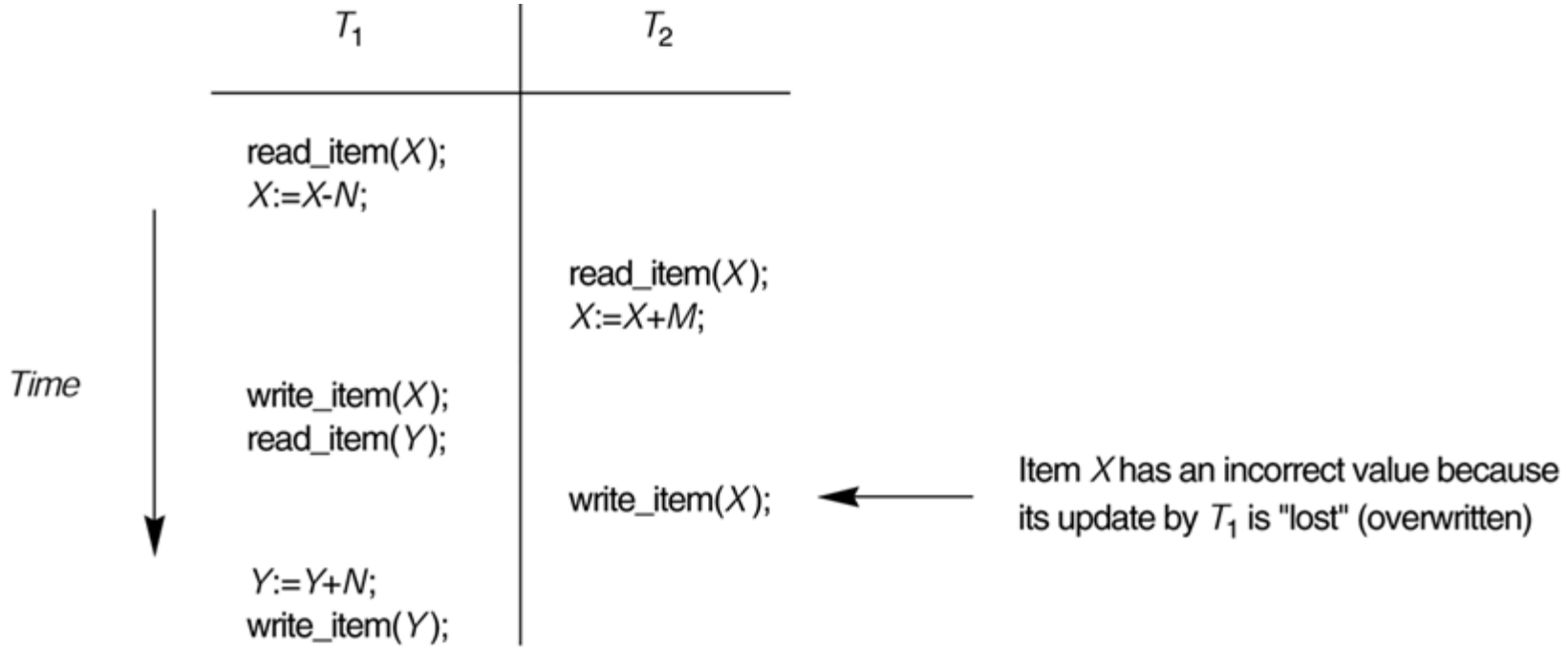
- **Concurrency control** is needed to achieve **isolation** i.e., to control the interaction among the concurrent transactions in order to prevent them from destroying the database **consistency**.

Why Concurrency Control is needed

- Advantages: increased processor and disk utilization; reduced average response time;
- Problems occur when concurrent transactions execute in an **uncontrolled** manner:
  - **The Lost Update**
  - **The Temporary Update (or Dirty Read)**
  - **The Incorrect Summary**
  - Unrepeatable Read:
    - A transaction T1 may read a given value. If another transaction later updates that value and T1 reads that value again, then T1 will see a different value.

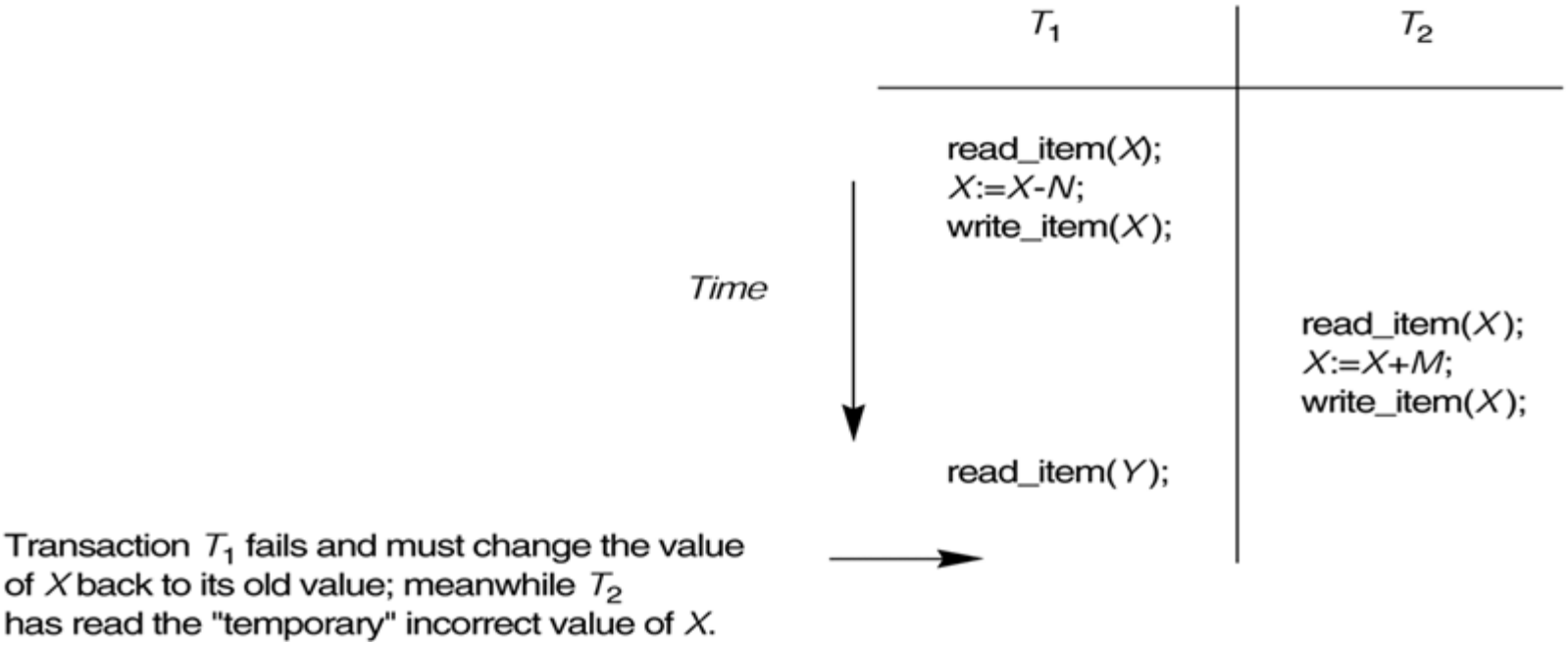
The Lost Update Problem

- This occurs when **two** transactions that access the **same** database items have their operations interleaved in a way that makes the value of some database item incorrect.



The Temporary Update Problem (Dirty Read)

- 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.

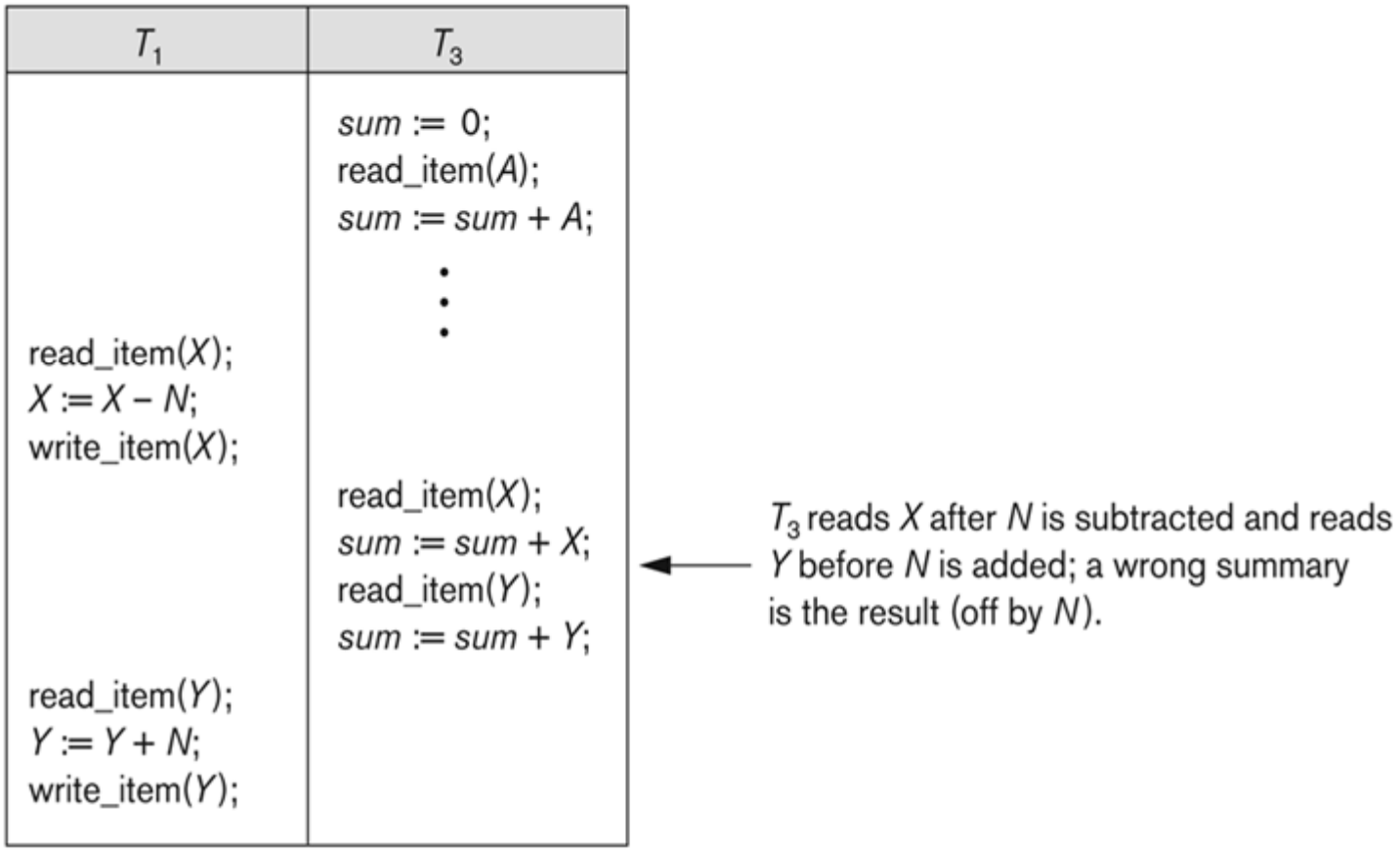


Transaction T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the "temporary" incorrect value of X.

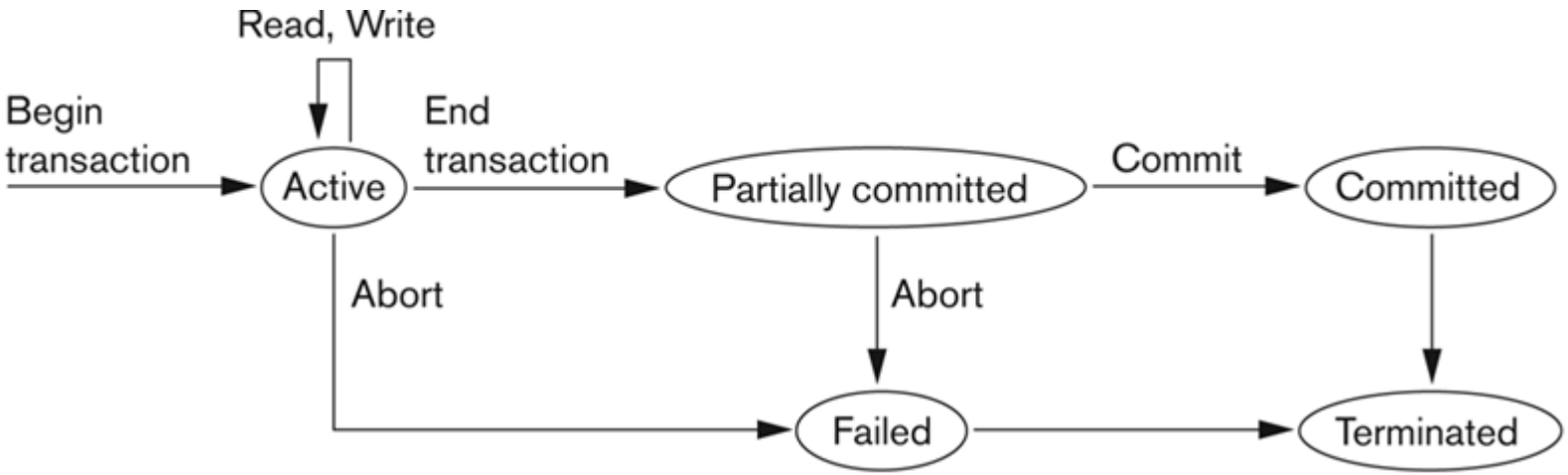
Incorrect Summary Problem

- 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.

Incorrect Summary Problem



Transaction State



Transaction state explanation

- 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**.
- A transaction must be in one of the following **states**:
  - **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.
  - **Committed:**
    - after *successful* completion.
  - **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 (only if no internal logical error)
      - **kill** the transaction

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.
  - 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*.

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 failure.
  - In addition, the log is periodically backed up to archival storage to guard against such failures.

The System Log

- **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.

Resources

- Chapter 21 of Fundamentals of Database Systems (FODS), 6th Edition.

Database Management System (15ECSC208)

UNIT II: Chapter 5: Transactions **(Part 2)**

Transaction Processing

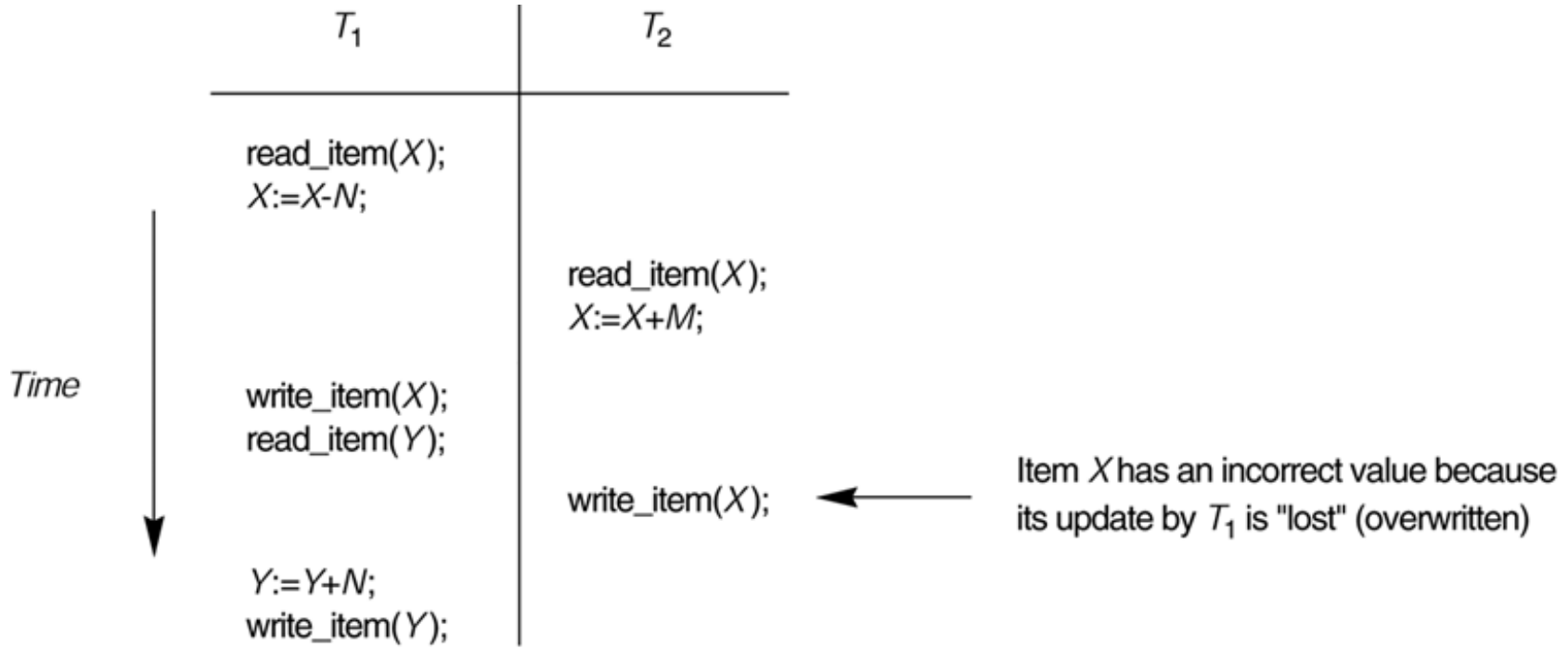
Schedules

- A ***schedule*** S of n transactions T_1, T_2, ..., T_n is an ordering of **all** the operations in these transactions subject to the constraint that:
  - for each transaction T_i, the operations of T_i in S must appear in the **same order** as they do in T_i.
    - Note, however, that operations from other transactions T_k can be interleaved with the operations of T_i in S.

- Example: Given
  - $T_1 = \mathbf{R}_1(Q)\ \mathbf{W}_1(Q)$ & $T_2 = \mathbf{R}_2(Q)\ \mathbf{W}_2(Q)$
  - a schedule: $\mathbf{R}_1(Q)\ \mathbf{R}_2(Q)\ \mathbf{W}_1(Q)\ \mathbf{W}_2(Q)$
  - not a schedule: $\mathbf{W}_1(Q)\ \mathbf{R}_1(Q)\ \mathbf{R}_2(Q)\ \mathbf{W}_2(Q)$

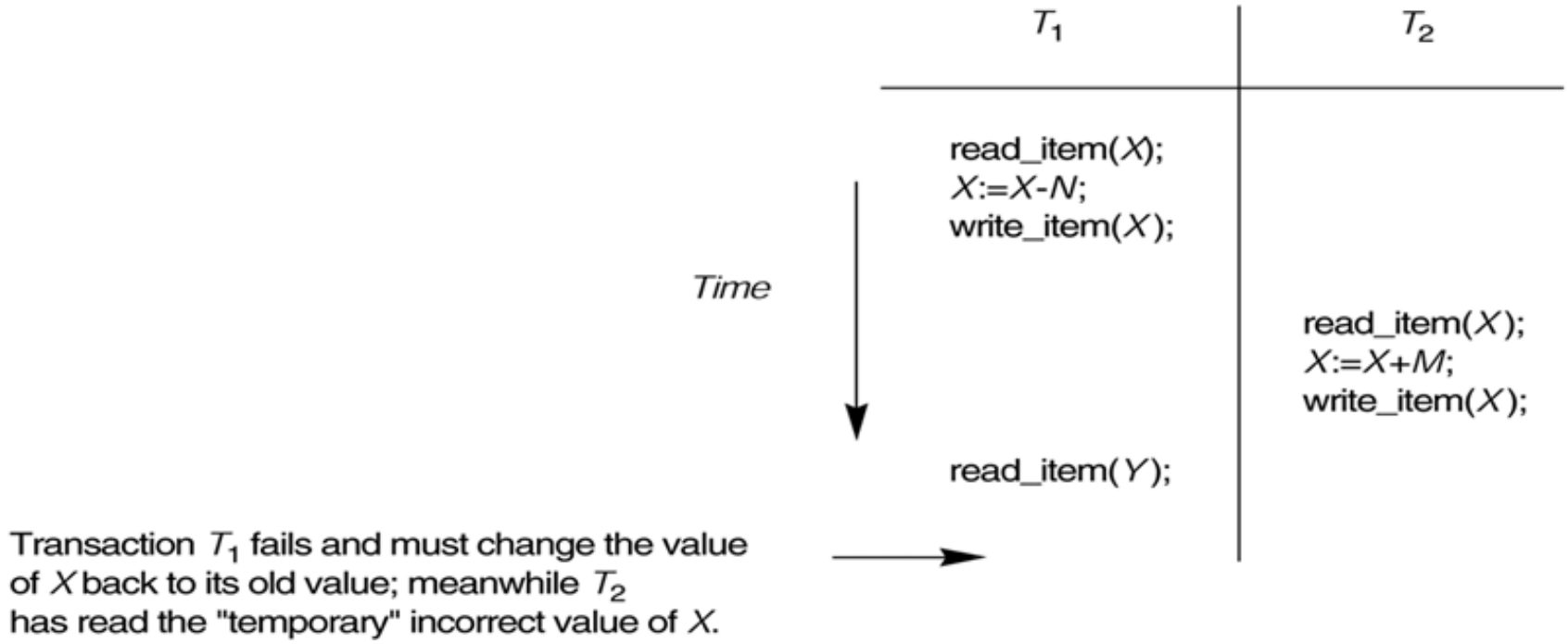
Schedules

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



Schedules

- S_b: $r_1(X)$; $w_1(X)$; $r_2(X)$; $w_2(X)$; $r_1(Y)$; a_1;



Conflict Operations

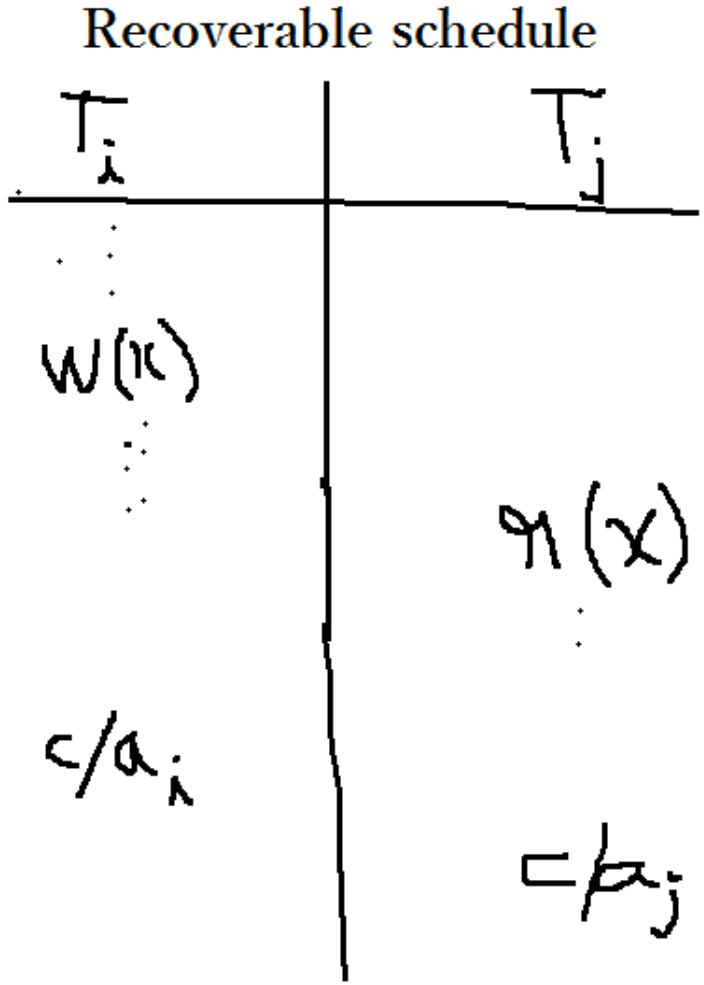
- Instructions (Operations) O_i and O_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item X accessed by both O_i and O_j, and at least one of these instructions is **write** X.

1. O_i = **Read**(X), O_j = **Read**(X). O_i and O_j don't conflict
2. O_i = **Read**(X), O_j = **Write**(X). They conflict
3. O_i = **Write**(X), O_j = **Read**(X). They conflict
4. O_i = **Write**(X), O_j = **Write**(X). They conflict

- Two operations in a schedule are **conflicting** if:
  1) They belong to **different** transactions,
  2) They access the same item X, and
  3) At least one of them is a **Write**(X) operation.

Recoverable Schedule

- **Def:** A schedule is recoverable if each transaction commits only after each transaction from which it has read has committed.
- **Def:** Schedule S is Recoverable if T_j reads a data item written by T_i, the COMMIT/ABORT operation of T_i appears before the COMMIT/ABORT operation of T_j
- ***Note:*** Once a transaction T is committed, it should never be necessary to rollback T.
- S_a, S_b and S_a' are recoverable:
  - S_a': $r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); c_2; w_1(Y); c_1;$
- Consider the following schedules:
  - S_c: $r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); c_2; a_1;$
  - S_d: $r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); c_1; c_2;$
  - S_e: $r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); a_1; a_2;$



S: ….$W_i(x)$….$r_j(x)$…….c/a_i…….c/a_j

Recoverable Schedule

Process:

- 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**.
- T **reads** from T' in S if X is first written by T' and later read by T.
  - T' should not have been aborted before T reads X
  - There should be no transaction T_i that writes X after T' writes it before T reads it (unless T_i, if any, has aborted before T reads X).

- S_c is not recoverable because:
  - T2 reads item X from T1, and then T2 commits before T1 commits.
  - If T1 aborts after c_2 operation in S_c, then the value of X that T2 read is no longer valid and T2 must be **aborted after** it is **committed**, leading to a schedule that is not **recoverable**.
  - For the schedule to be recoverable c_2 operation in S_c must be postponed until after T1 commits, as shown in Sd;
  - If T1 aborts instead of committing, then T2 should also abort as shown in Se, because the value of X it read is no longer valid.

Cascading Rollback

- **Cascading rollback/ Cascading abort:**
  - A schedule in which **uncommitted** transactions has to be **rolled back** because it **read** an item from a transaction that failed.
    - As shown in schedule S_e

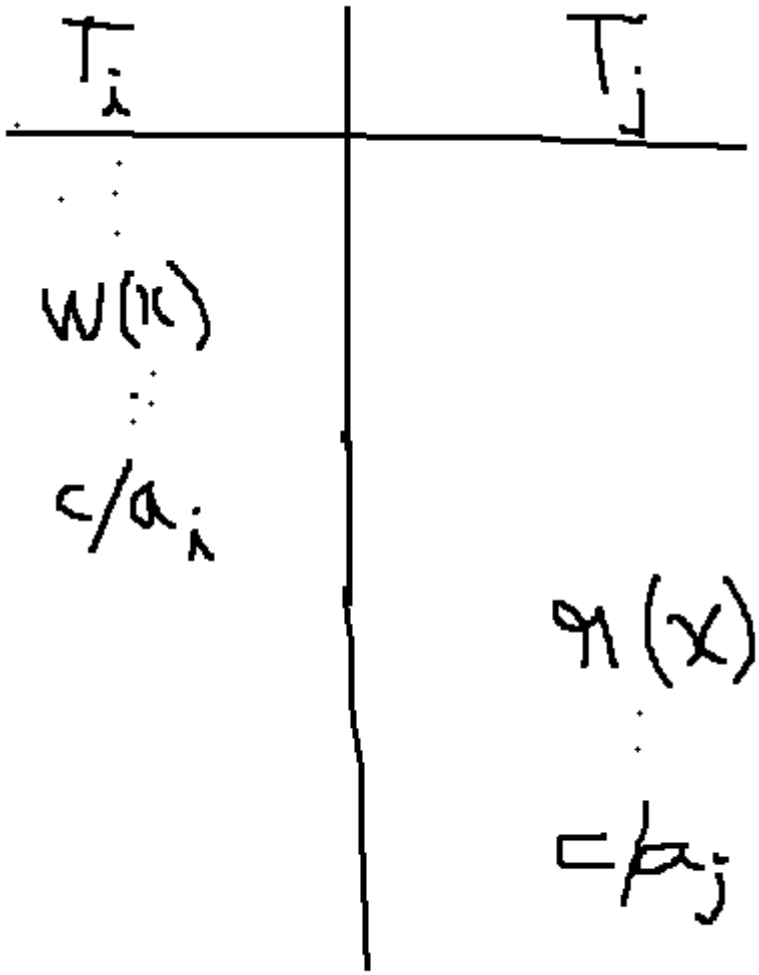
- A single transaction abort leads to a series of transaction Rollback.

- **Q:** What is the main cause of cascading rollback?
- **Ans: DIRTY READ**: reading an output of an uncommitted transaction.

Avoids Cascading Rollback (ACR) Schedule

- **Def:** A schedule is ACR, if transactions may read only values written by committed transactions.
- **Def:** Schedule S is CASCADELESS if T_j reads a data item written by T_i, the COMMIT/ABORT operation of T_i appears before the READ operation of T_j.
- ***Note:*** One where every transaction **reads** only the items that are **written** by committed transactions.
  - S_d: $r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); c_1; c_2;$
  - S_e: $r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); a_1; a_2;$
- Also know as **Cascadeless Schedule**.

Cascadeless/ACR schedule

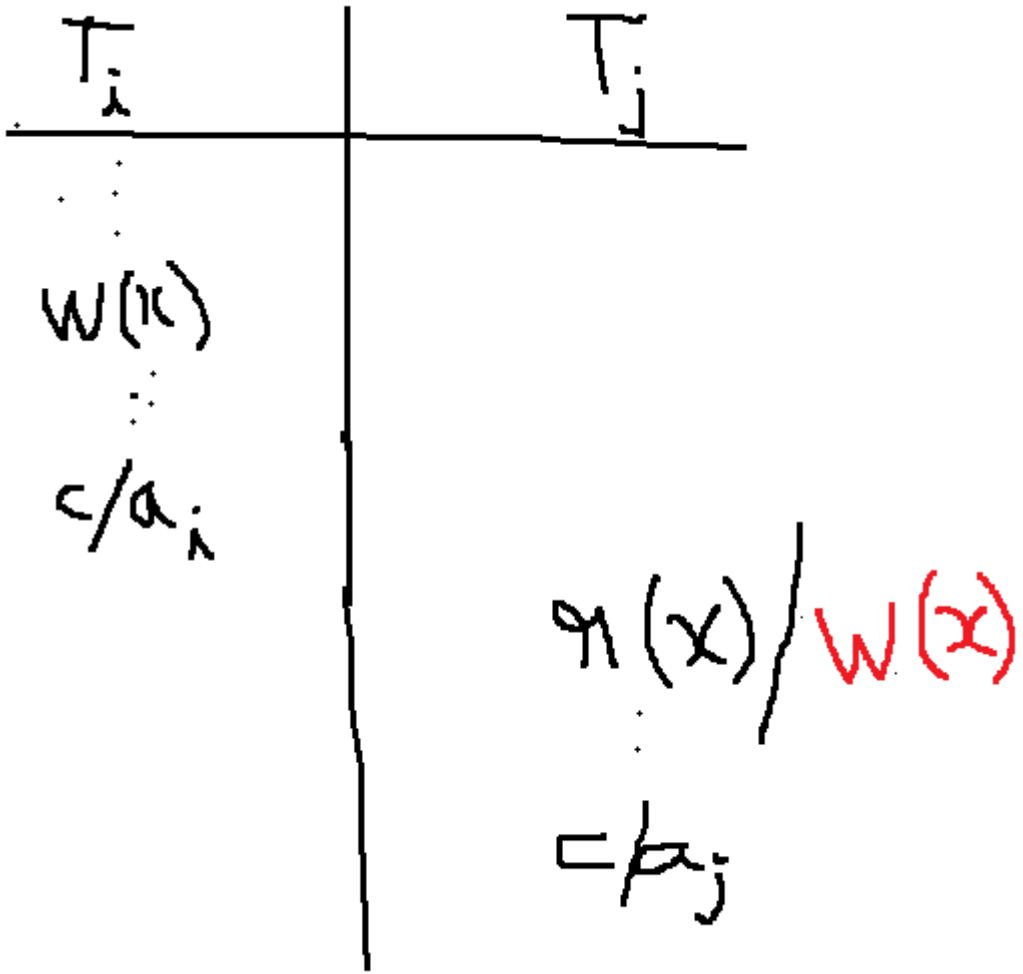


S: ….$W_i(x)$…. c/a_i…… $r_j(x)$……. ….c/a_j

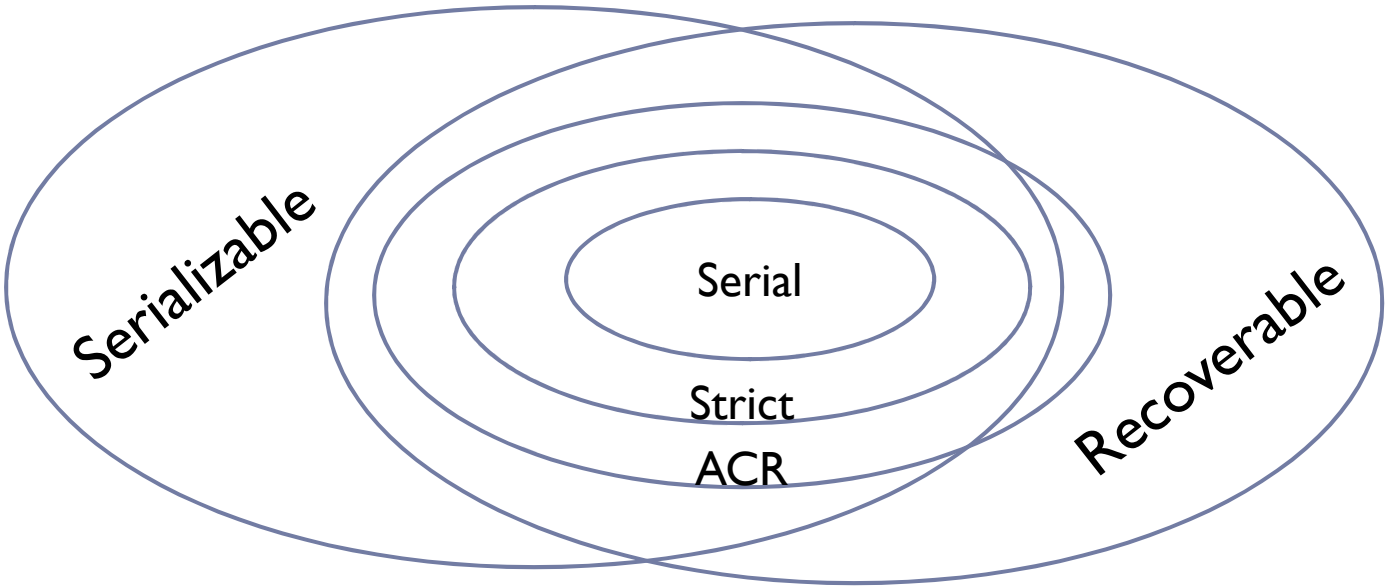
Strict Schedule

- **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.
- Consider the following schedule:
  - S_f: $w_1(X, 5)$; $w_2(X, 8)$; a_1;
- Although S_f is cascade-less, it is not strict
  - It permits T_2 to write X even though T_1 that last wrote X had not yet committed (or aborted).

Strict schedule



Relationships among different schedules



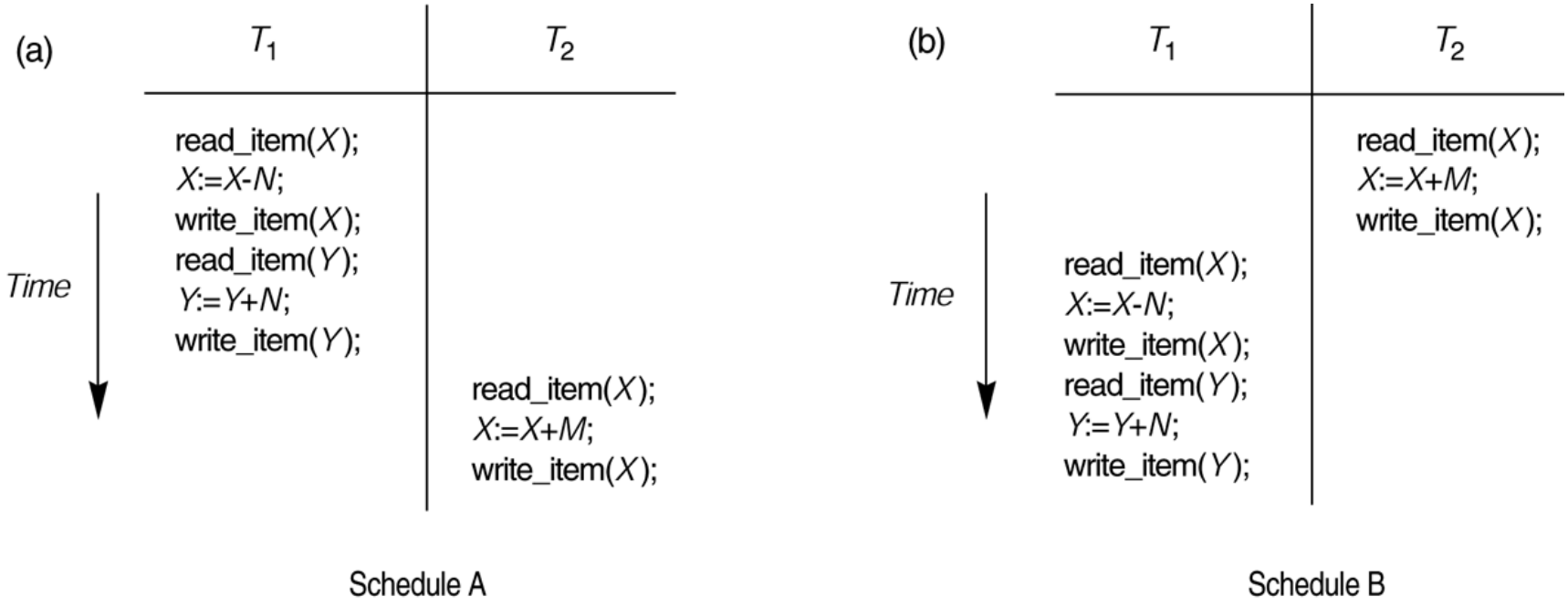
- Every ACR is Recoverable.
- Every strict schedule is ACR.
- Every strict schedule is serializable.

Serial Schedules

- A schedule *S* is **serial**,
  - if for every transaction *T* participating in the schedule, all the operations of *T* are executed ***consecutively***
    - if operations from different transactions are **not** interleaved.
  - otherwise the schedule is called ***non-serial***.
- **Complete Schedule:** A schedule that contains either a commit /abort action for EACH transaction.
- Serial schedules:
  - $\mathbf{R}_1(Q)\ \mathbf{W}_1(Q)\ \mathbf{R}_2(Q)\ \mathbf{W}_2(Q)$
  - $\mathbf{R}_2(Q)\ \mathbf{W}_2(Q)\ \mathbf{R}_1(Q)\ \mathbf{W}_1(Q)$
- Non-serial schedule:
  - $\mathbf{R}_1(Q)\ \mathbf{R}_2(Q)\ \mathbf{W}_1(Q)\ \mathbf{W}_2(Q)$

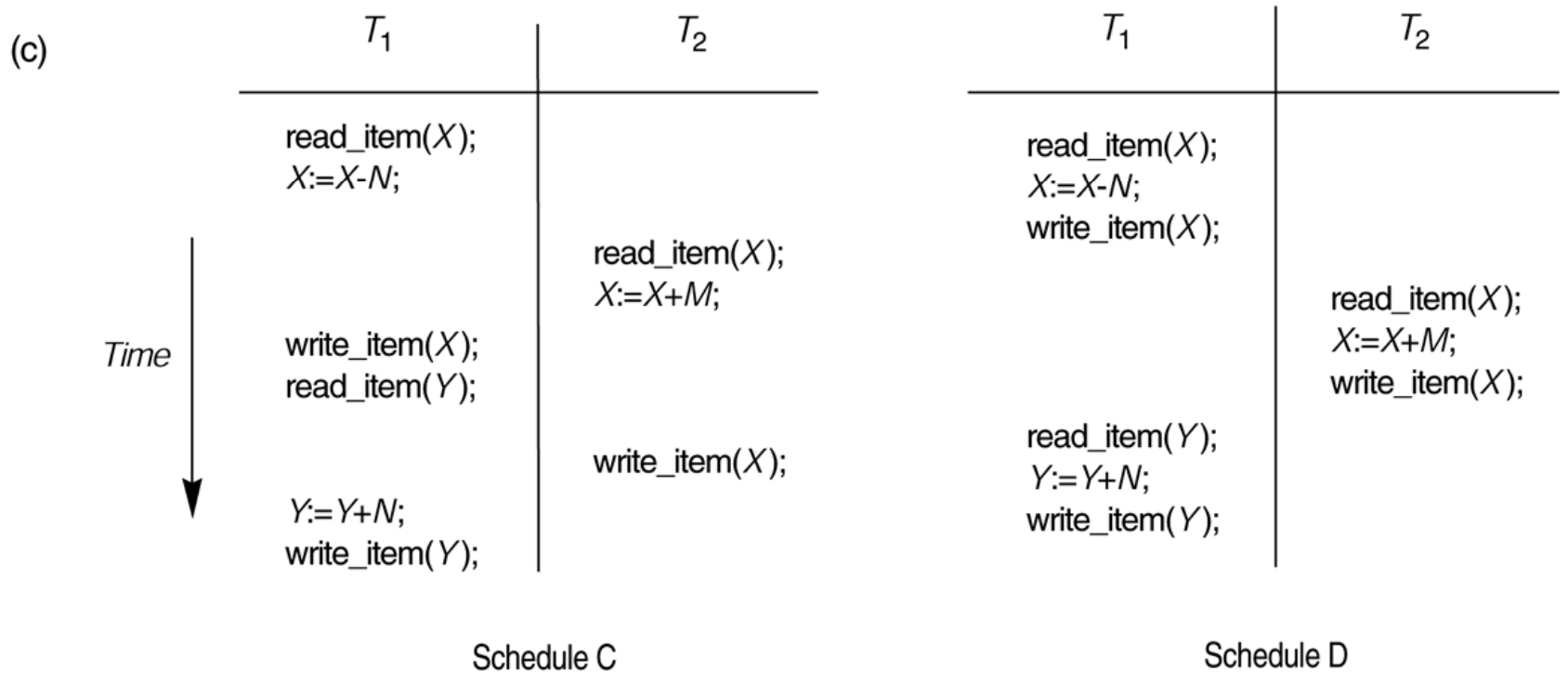
Example Serial Schedules

- (a) Serial schedule A: T1 followed by T2.
- (b) Serial schedules B: T2 followed by T1.



Example non-serial Schedule

- (c) Two **nonserial** schedules C and D with interleaving of operations.



Several Observations

- Serial schedule guarantees database consistency.
- *n* transactions may form *n*! different serial schedules.
- Allowing only serial schedule may cause poor system performance (i.e., low throughput)
- Serial schedule is **not** a must for guaranteeing transaction consistency.

Serializability

- One way to ensure correctness of concurrent transactions is to enforce **serializability** of transactions
  - that is the interleaved execution of the transactions must be **equivalent** to some serial execution of those transactions.

- **Definition1**: A schedule is said to be **serializable** if the result of executing that schedule is the same as the result of executing ***some*** serial schedule.
- **Definition2**: A schedule *S* of *n* transactions is **serializable** if it is ***equivalent to some serial schedule*** of the same *n* transactions.

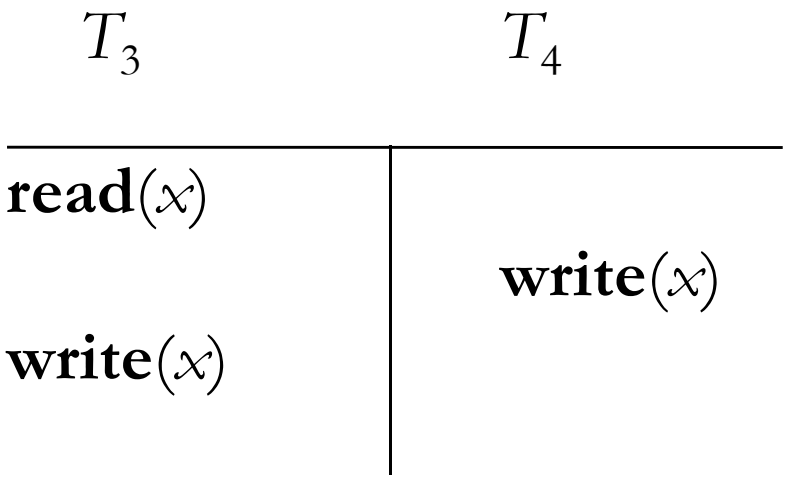
- Different forms of **schedule equivalence**:
  - Conflict equivalence (conflict serializability)
  - View equivalence (view serializability)

Conflict Serializability

- Two schedules are said to be **conflict equivalent** if the order of any two conflicting operations is the same in both schedules.
- We say that a schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule
- Example: Consider two transactions:
  - $T_1 = \mathbf{R}_1(X)\ \mathbf{W}_1(X)\ \mathbf{R}_1(Y)\ \mathbf{W}_1(Y)$
  - $T_2 = \mathbf{R}_2(X)\ \mathbf{W}_2(X)$
- The following two schedules are equivalent:
  - $\mathbf{S}_1$: $\mathbf{R}_1(X)\ \mathbf{W}_1(X)\ \mathbf{R}_2(X)\ \mathbf{W}_2(X)\ \mathbf{R}_1(Y)\ \mathbf{W}_1(Y)$
  - $\mathbf{S}_2$: $\mathbf{R}_1(X)\ \mathbf{W}_1(X)\ \mathbf{R}_1(Y)\ \mathbf{W}_1(Y)\ \mathbf{R}_2(X)\ \mathbf{W}_2(X)$

Conflict Serializability (Cont.)

- Example of a schedule that is **not conflict serializable**:



[r3(x); w4(x); w3(x)]

- Serial schedule - <T_3, T_4> or <T_4, T_3>
  - T3, T4: r3(x); w3(x); w4(x)
  - T4,T3: w4(x); r3(x);w3(x)

Testing for Conflict Serializability Schedules

Algorithm Precedence_Graph()

{

Step-1: Draw nodes corresponding to each of the transaction in S.

Step-2: Draw an edge from Ti to Tj, if Ti precedes and conflicts with Tj. Complete the precedence graph. (The conflicting actions are RW, WR, and WW).

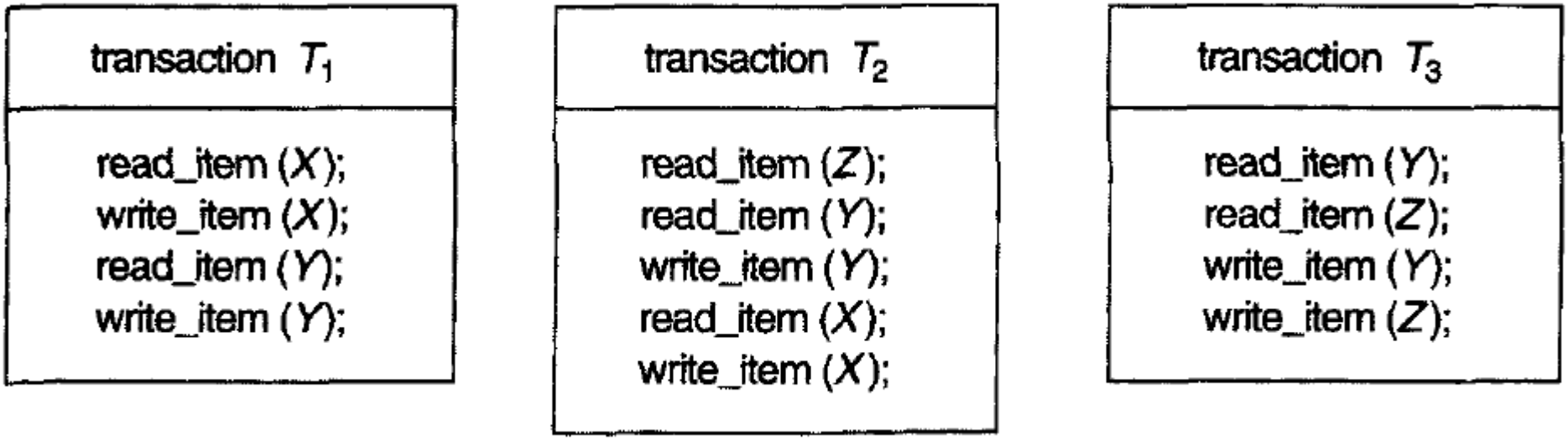
Step-3: If the precedence graph is acyclic,

then print "S is conflict serializable".

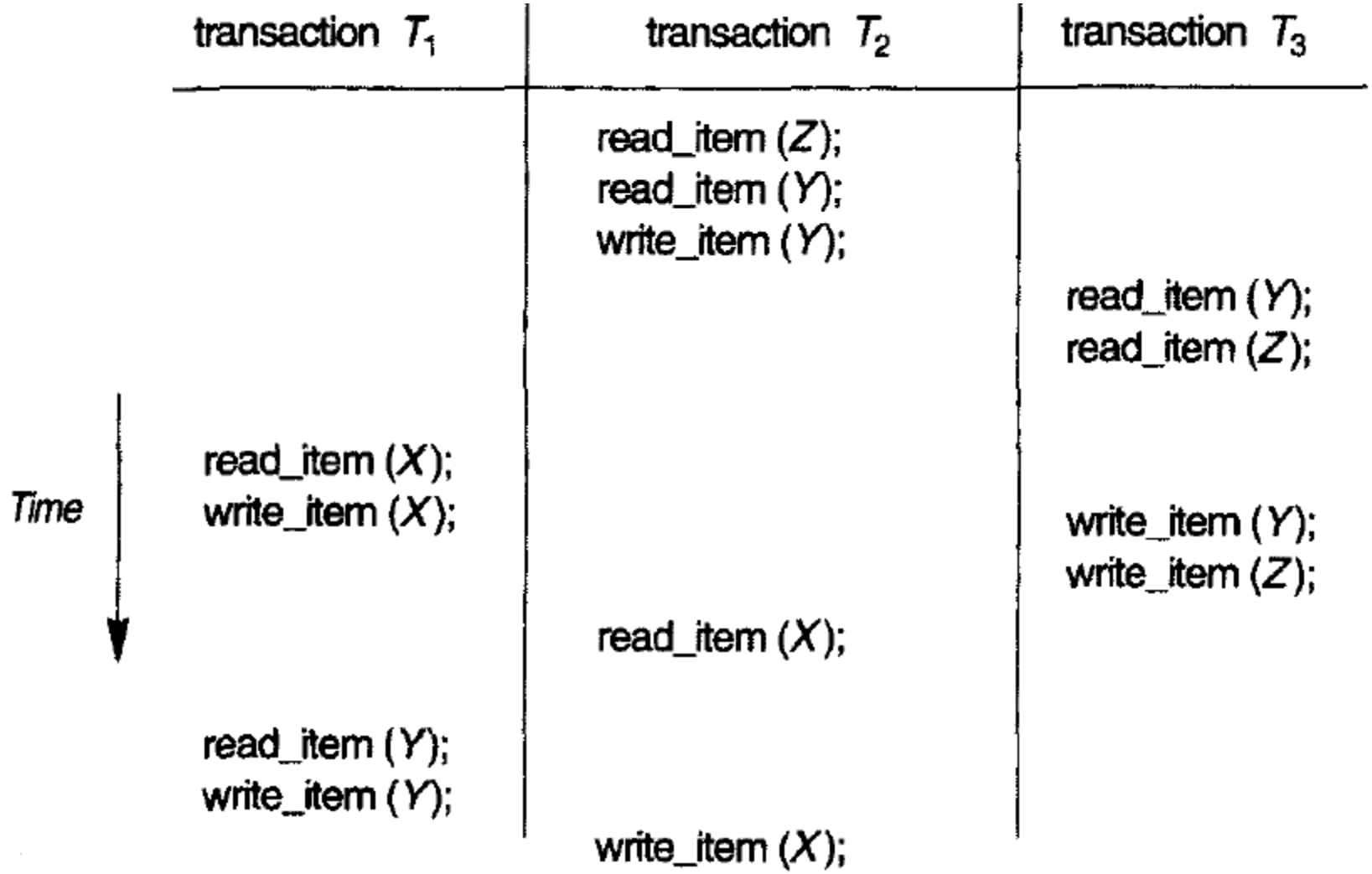
else print "S is not conflict serializable".

}

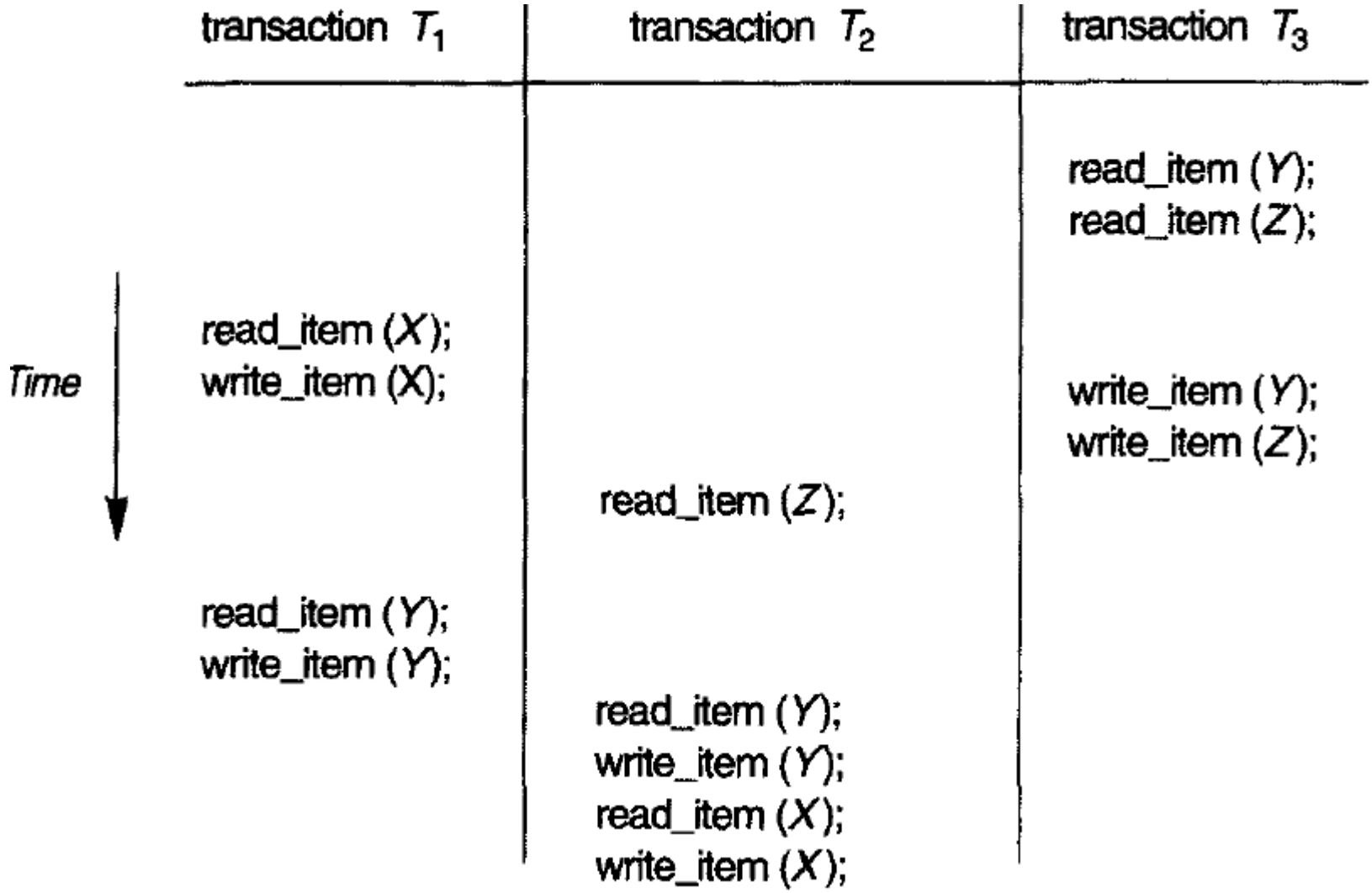
Example on Serializability testing



Example1 on Serializability testing

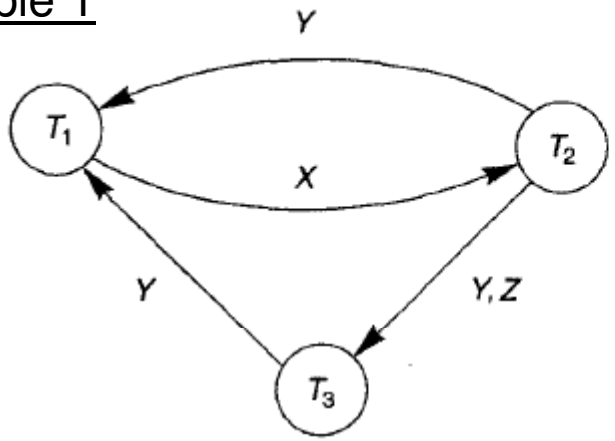


Example2 on Serializability testing



Example on Serializability testing

Example 1



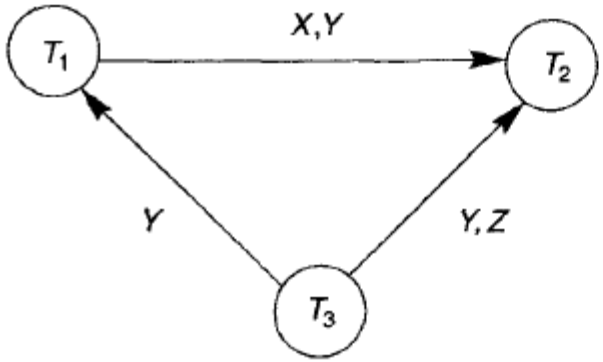
Equivalent serial schedules

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 2



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

View Serializability

- Let *S1* and *S2* be two schedules with the same set of transactions. *S1* and *S2* are **view equivalent** if the following three conditions are met:

1. If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2
2. If Tj reads value of A written by Ti in S1, then Tj also reads value of A written by Ti in S2
3. If Ti writes final value of A in S1, then Ti also writes final value of A in S2

View Serializability

- As can be seen, *view equivalence* is based purely on **reads** and **writes** alone.

- Conditions 1 and 2 ensure that
  - each transaction reads the same values in both schedules.
- Condition 3, coupled with conditions 1 and 2, ensures that
  - both schedules results in the same final state

View Serializability

- A schedule *S* is **view serializable**
  - if it is *view equivalent* to a serial schedule.
- Every *conflict serializable* schedule is also *view serializable*.
- Below schedule is **view-serializable** but **not** conflict serializable.
- Every *view serializable* schedule that is **not** conflict serializable has ***blind writes***.
  - a write operation without having performed a read operation

T1	T2	T3
read(x)		
	write(x)	
write(x)		
		write(x)

Summary: Schedules

- Schedules must be **conflict** or **view** serializable, and **recoverable** - for the sake of database consistency, and preferably **cascadeless**.

Resources

- Chapter 21 of Fundamentals of Database Systems (FODS), 6th Edition.