Database Management Systems

ITC 304

Subject Incharge

Priyanka Patil

Associate Professor

Room No. 316

email: priyankapatil@sfit.ac.in

Module VI

Lecture 1

Transactions Management and Concurrency and Recovery

Content

Introduction to Transaction Processing

- **Concurrency**

  Interleaved processing:

  Concurrent execution of processes is interleaved in a single CPU

  Parallel processing:

  Processes are concurrently executed in multiple CPUs.

Interleaved processing vs Parallel processing

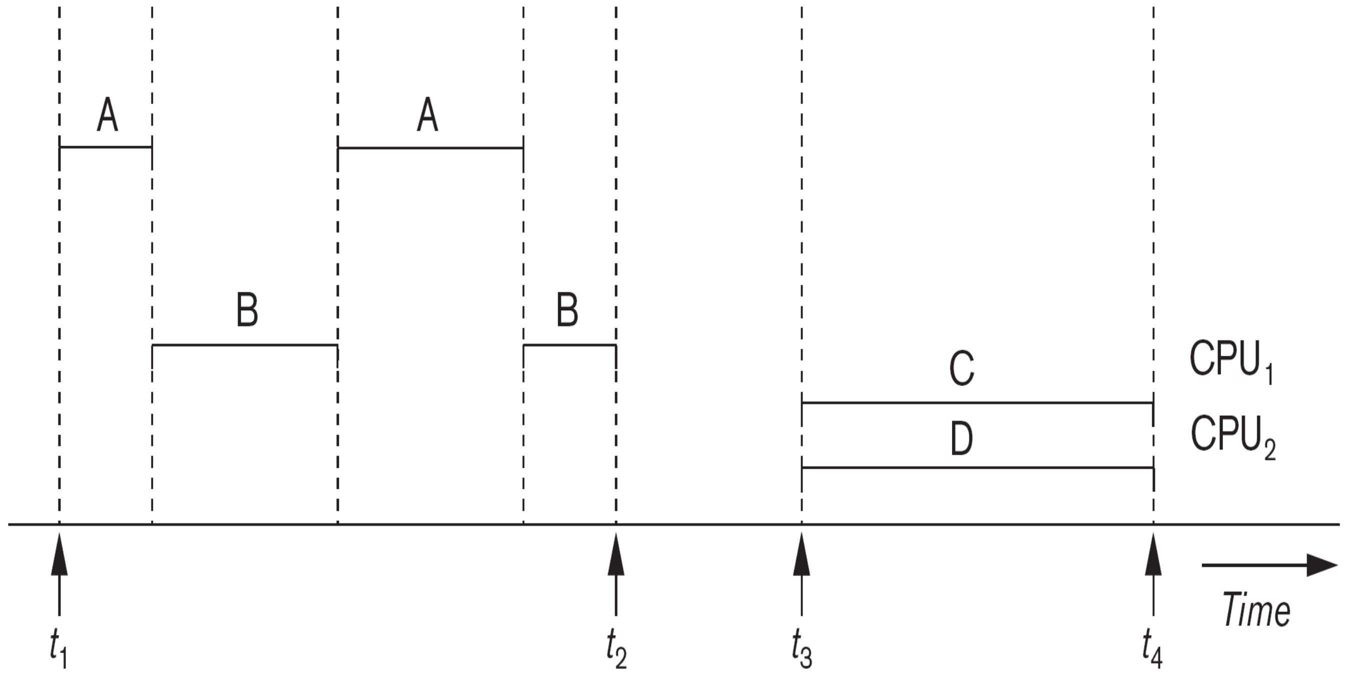


Figure 21.1
Interleaved processing versus parallel processing of concurrent transactions.

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).

  It is a set of operations used to perform a logical unit of work.

- **Transaction boundaries**:

  Begin and End transaction.
- An **application program** may contain several transactions separated by the Begin and End transaction boundaries.

- Example of Transaction:

T1

Read(X)

Introduction to Transaction Processing

READ AND WRITE OPERATIONS:

Basic unit of data transfer from the disk to the computer main memory is one block. In general, a data item (what is read or written) will be the field of some record in the database, although it may be a larger unit such as a **record** or even a **whole block.**

read_item(X) command includes the following **steps**:

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 buffer to the program variable named X.

READ AND WRITE OPERATIONS (contd.):

write_item(X) command includes the following steps:

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

Desirable Properties of Transaction or ACID Properties

A -

C -

I -

D -

Desirable Properties of Transaction or ACID Properties

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 (see Chapter 21).
- **Durability or permanency**: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

Example of Fund Transfer

1.Atomicity

- It states that all operations of the transaction take place at once if not, the transaction is aborted.
- There is no midway, i.e., the transaction cannot occur partially. Each transaction is treated as one unit and either run to completion or is not executed at all.
- Atomicity involves the following two operations:
- **Abort:** If a transaction aborts then all the changes made are not visible.
- **Commit:** If a transaction commits then all the changes made are visible.

Example of Fund Transfer

- **Example:** Let's assume that following transaction T consisting of T1 and T2. A consists of Rs 600 and B consists of Rs 300. Transfer Rs 100 from account A to account B.

T1	T2
Read(A)	Read(B)
A:= A-100	Y:= Y+100
Write(A)	Write(B)

- After completion of the transaction, A consists of Rs 500 and B consists of Rs 400.
- If the transaction T fails after the completion of transaction T1 but before completion of transaction T2, then the amount will be deducted from A but not added to B. This shows the inconsistent database state. In order to ensure correctness of database state, the transaction must be executed in entirety.

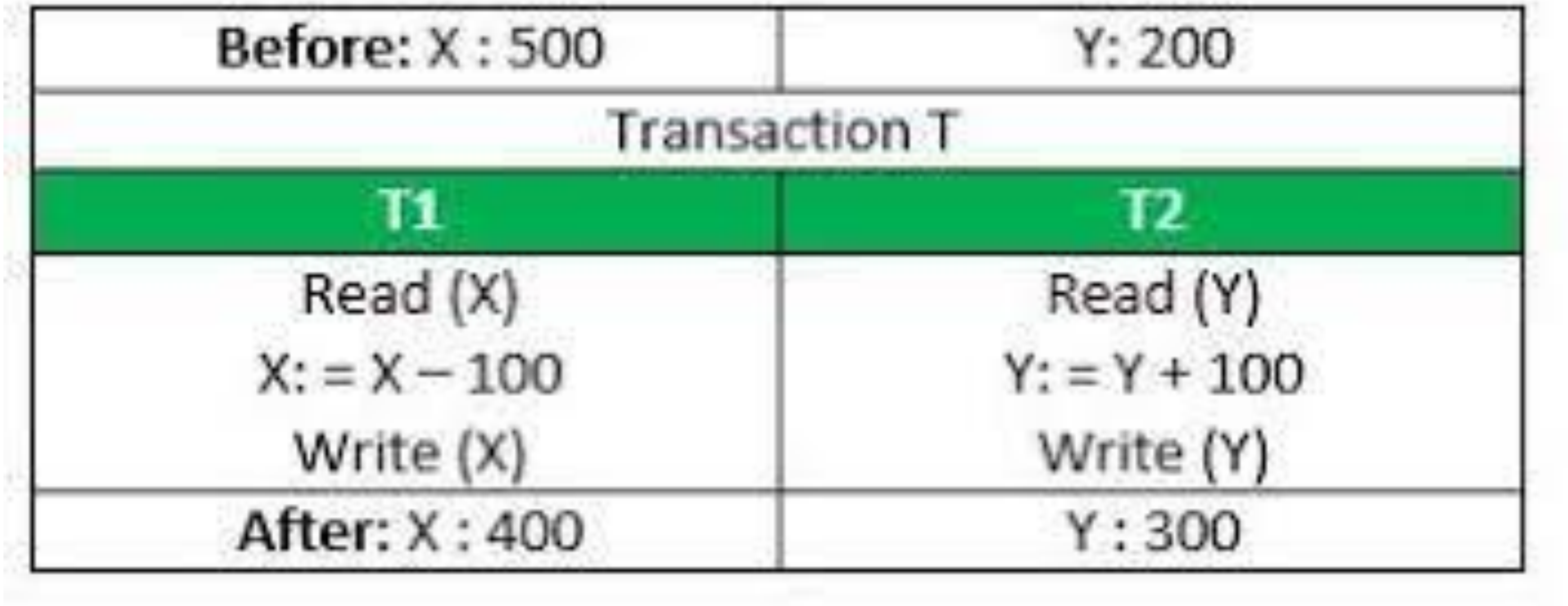
Example of Fund Transfer

2.Consistency

- The **integrity constraints** are maintained so that the database is **consistent before and after the transaction.**
- The execution of a transaction will leave a database in either its prior stable state or a new stable state.
- The consistent property of database states that every transaction sees a consistent database instance.
- The transaction is used to transform the database from one consistent state to another consistent state.

Example of Fund Transfer

- **For example:** The total amount must be maintained before or after the transaction.



Before: X : 500	Y: 200
Transaction T	
T1	T2
Read (X) X: = X – 100 Write (X)	Read (Y) Y: = Y + 100 Write (Y)
After: X : 400	Y : 300

Example of Fund Transfer

3.Isolation

- It shows that **the data** which is used at the time of execution of a transaction cannot be used by the **second transaction** until the first one is completed.
- In isolation, **if the transaction T1 is being executed** and using the data item X, then that data item **can't be accessed by any other transaction T2** until the transaction T1 ends.
- The concurrency control subsystem of the DBMS enforced the isolation property.

Example of Fund Transfer

4.Durability

- The durability property is used to indicate the performance of the database's c**onsistent state**. It states that the transaction made the **permanent changes.**
- They cannot be lost by the erroneous operation of a faulty transaction or by the system failure. When a transaction is completed, then the database reaches a state known as the consistent state. That consistent state cannot be lost, even in the event of a system's failure.
- The recovery subsystem of the DBMS has the responsibility of Durability property.

Transaction and System Concepts

- A **transaction** is an atomic unit of work that is either completed in its entirety or not done at all.
  - For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts.
- **Transaction states**:
  - Active state
  - Partially committed state
  - Committed state
  - Failed state
  - Terminated State

Transaction States

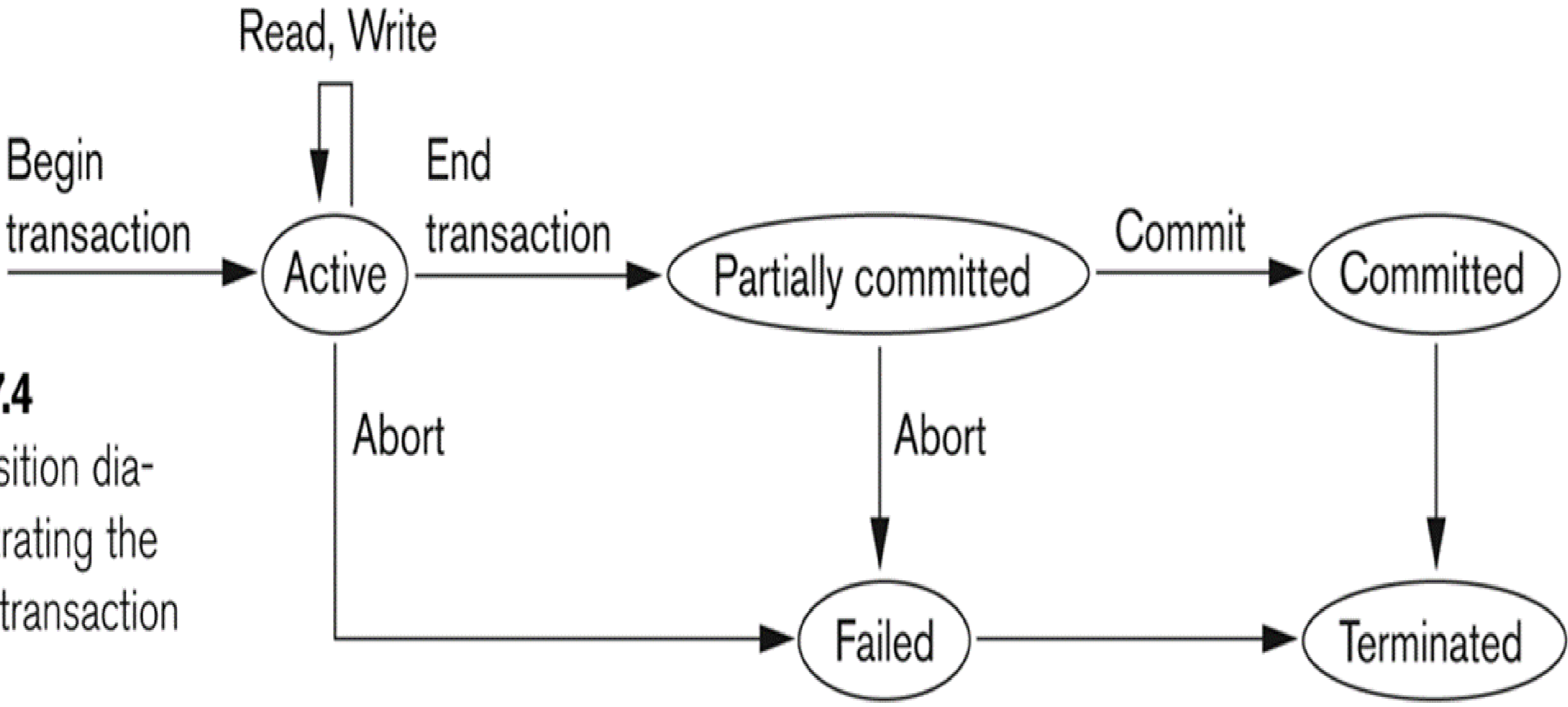


Figure 17.4
State transition diagram illustrating the states for transaction execution.

Transaction States

1. Active state

- The active state is the first state of every transaction. In this state, the transaction is being executed.
- For example: Insertion or deletion or updating a record is done here. But all the records are still not saved to the database.

2. Partially committed

- In the partially committed state, a transaction executes its final operation, but the data is still not saved to the database.
- In the total mark calculation example, a final display of the total marks step is executed in this state.

Transaction States

3. Committed

- A transaction is said to be in a committed state if it executes all its operations successfully. In this state, all the effects are now permanently saved on the database system.

4. Failed state

- If any of the checks made by the database recovery system fails, then the transaction is said to be in the failed state.
- In the example of total mark calculation, if the database is not able to fire a query to fetch the marks, then the transaction will fail to execute.

Transaction States

5.**Aborted**

- If any of the checks fail and the transaction has reached a **failed state** then the database recovery system will make sure that the database is in its previous consistent state. If not then it will abort or roll back the transaction to bring the database into a consistent state.
- If the transaction fails in the middle of the transaction then before executing the transaction, all the executed transactions are rolled back to its consistent state.
- After aborting the transaction, the database recovery module will select one of the two operations:
  1. Re-start the transaction
  2. Kill the transaction

Transaction Control Commands

- The Transaction Control Language **manages the changes** in the data which are made by the DML operations.
- The alteration made by the DML commands such as UPDATE, INSERT or DELETE is not permanent and these changes can be canceled before the present session gets closed.
- To control the changes and processing of data, TCL is used simultaneously with the Data Manipulation Language.
- As we perform many operations on the data, the database might become inconsistent between the transactions.
- So the Transaction Control Language (TCL) is used to **maintain consistency and manage the transaction** in a database.
- After the commit is performed the database state is changed from one to another consistent state.
- The Transactions are used on all the DDL and DML queries automatically.

Transaction Control Commands(Cont…)

- **1. COMMIT**
- This command is used to make a transaction permanent in a database. So it can be said that commit command saves the work done as it ends the current transaction by making permanent changes during the transaction.
- syntax :

```
COMMIT;
```

Transaction Control Commands(cont..)

- For instance, we want to update the location of an employee in the table “EMPLOYEE”.
- Let us update the EMP_ LOC for Raju as below:
- **Query:**

```
UPDATE EMPLOYEE SET EMP_ LOC = 'Hyderabad' WHERE
EMP_NAME= 'Raju';

COMMIT;
```

***The update transaction is completed with the commit command as above and the usage of the above statements will update the location of the employee ‘Raju’ and the change will be saved in the database permanently.

Transaction Control Commands(Cont..)

- ***ROLLBACK :***
- The ROLLBACK command is the transactional command used to undo transactions that have not already been saved to the database.

- **syntax :**

```
ROLLBACK;
```

- Also, the ROLLBACK command is used along with savepoint command to leap to a save point in a transaction.
- syntax :

```
ROLLBACK TO <savepoint_name>;
```

Transaction Control Commands(Cont..)

- Let us consider that we have updated EMP_LOC for Raju to Bangalore later and realize that the update was done mistakenly as below. Then we can restore the EMP_LOC for 'Raju' to Hyderabad again by using the Rollback command as below.

- **Query:**

```
UPDATE EMPLOYEE SET EMP_LOC= 'Bangalore' WHERE
EMP_NAME = 'Raju';
ROLLBACK;
```

Transaction Control Commands(Cont..)

3. SAVEPOINT

- This command is used to save the **transaction temporarily.** So the users can rollback to the required point of the transaction.
- syntax :

```
SAVEPOINT savepoint_name;
```

The System Log

Log or Journal:

The log keeps track of all transaction operations that affect the values of database items.

- This information may be needed to permit recovery from **transaction failures.**
- The log is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure.
- In addition, the log is **periodically backed up** to archival storage (tape) to guard against such catastrophic failures.

The System Log(cont)

- **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.

The System Log(cont)

- **[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.**

Schedules

A sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed.

- A schedule for a set of transactions must consist of **all instructions** of those transactions
- Must preserve the **order** in which the instructions appear in each individual transaction.

By default transaction assumed to execute **commit** instruction as its last step.

A transaction that fails to successfully complete its execution will have an **abort instruction** as the last statement

Schedules

There are two types of Schedules:

1. Serial Executions(Schedule)
2. Concurrent Executions(Schedule)

Schedule 1

- Let T_1 transfer \$50 from *A* to *B*, and T_2 transfer 10% of the balance from *A* to *B.*
- A serial schedule in which T_1 is followed by T_2 :

T_1	T_2
read (*A*)	
$A := A - 50$	
write (*A*)	
read (*B*)	
$B := B + 50$	
write (*B*)	
commit	
	read (*A*)
	$temp := A * 0.1$
	$A := A - temp$
	write (*A*)
	read (*B*)
	$B := B + temp$
	write (*B*)
	commit

Schedule 2

- A serial schedule where T_2 is followed by T_1

T_1	T_2
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	

Concurrent Executions

- ❑ In a multi-user system, multiple users can access and use the **same database at one time**, which is known as the **concurrent execution** of the database. It means that the same database is executed simultaneously on a multi-user system by **different users**.
- ❑ No operation should affect the other executing operations, thus **maintaining the consistency** of the database.
- ❑ Thus, on making the concurrent execution of the transaction operations, there occur several challenging **problems** that need to be solved.

Schedule 3

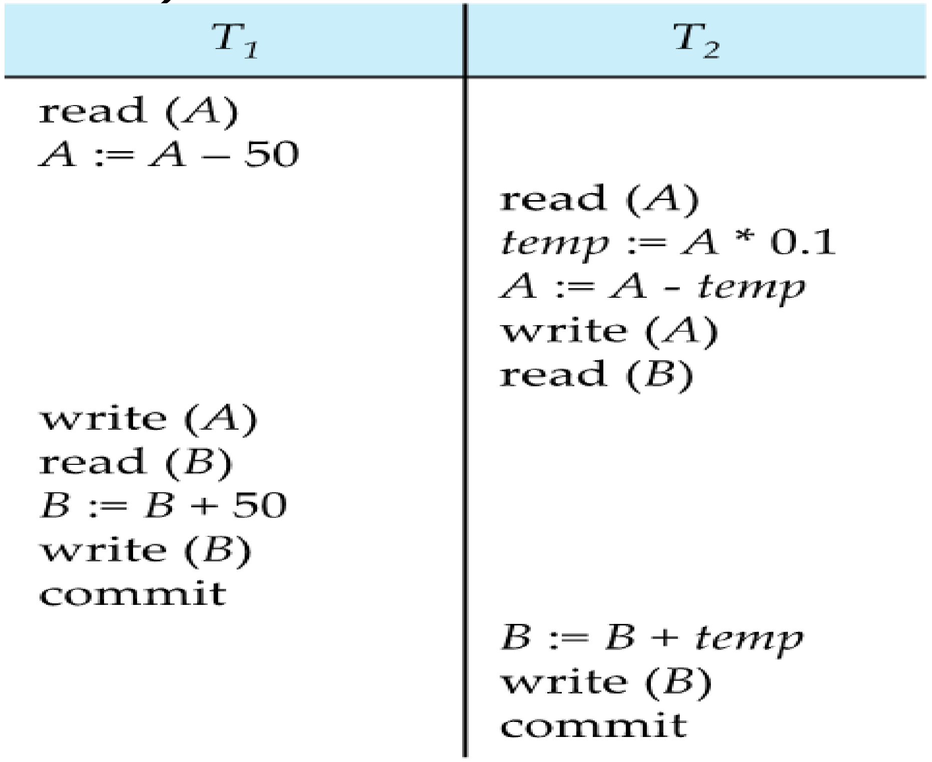
- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

In Schedules 1, 2 and 3, the sum A + B is preserved.

Schedule 4

The following concurrent schedule **does not preserve** the value of (*A* + *B*).



T_1	T_2
read (A)	
$A := A - 50$	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	$B := B + temp$
	write (B)
	commit

Problems with Concurrent Execution of Transaction

Problem occurs when two transactions are **dealing with the same data item** and if **one** of the transactions perform **write operation.**

1. Temporary Update Problem(Dirty Read Problem)
2. Incorrect Summary Problem
3. Lost Update Problem
4. Unrepeatable Read Problem

Problems with Concurrent Execution of Transaction

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.

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.

Problems with Concurrent Execution of Transaction

The Lost Update Problem(Write-Write Conflicts)

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.**

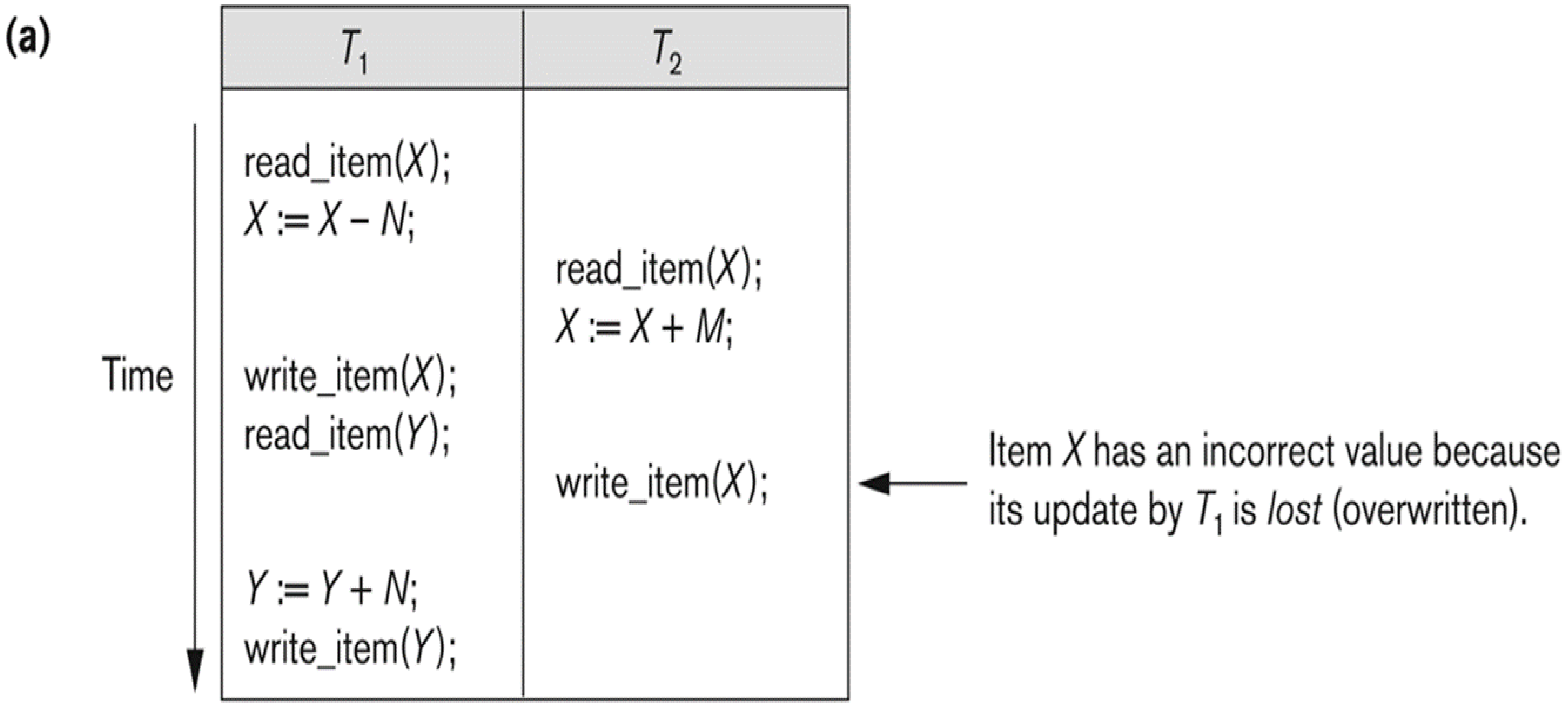
The Unrepeatable Read Problem

It one transaction **T1 reads** the **same item twice** and the item is **changed by another transaction** T2 in between two reads. Then T1 will **receive different values** for its two reads of the **same item.**

Problem 1: Lost Update Problems (W - W Conflict)

Figure 17.3

Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.



(a)

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

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

Problem 2: The temporary update problem.

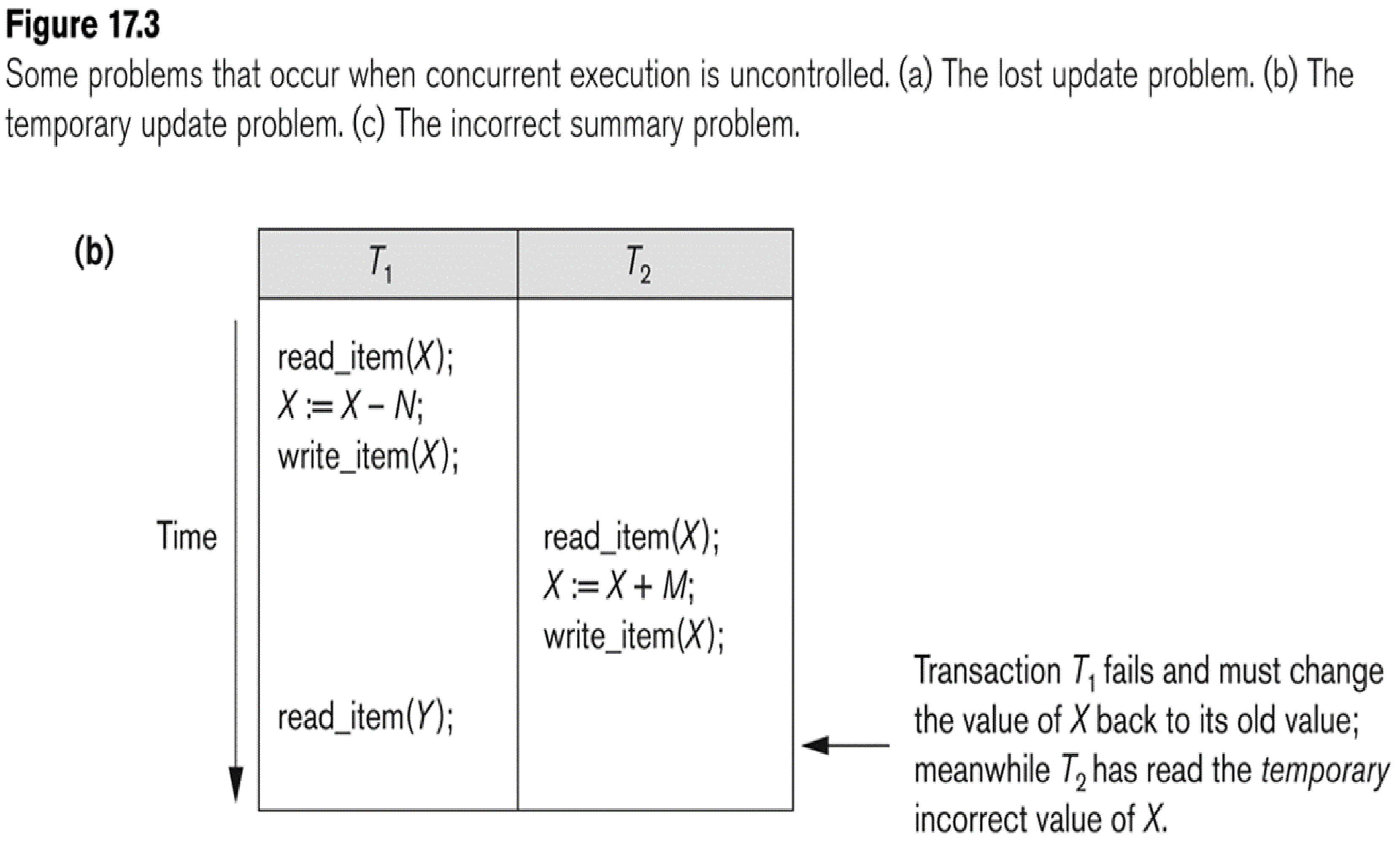


Figure 17.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

(b)

T_1	T_2

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

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

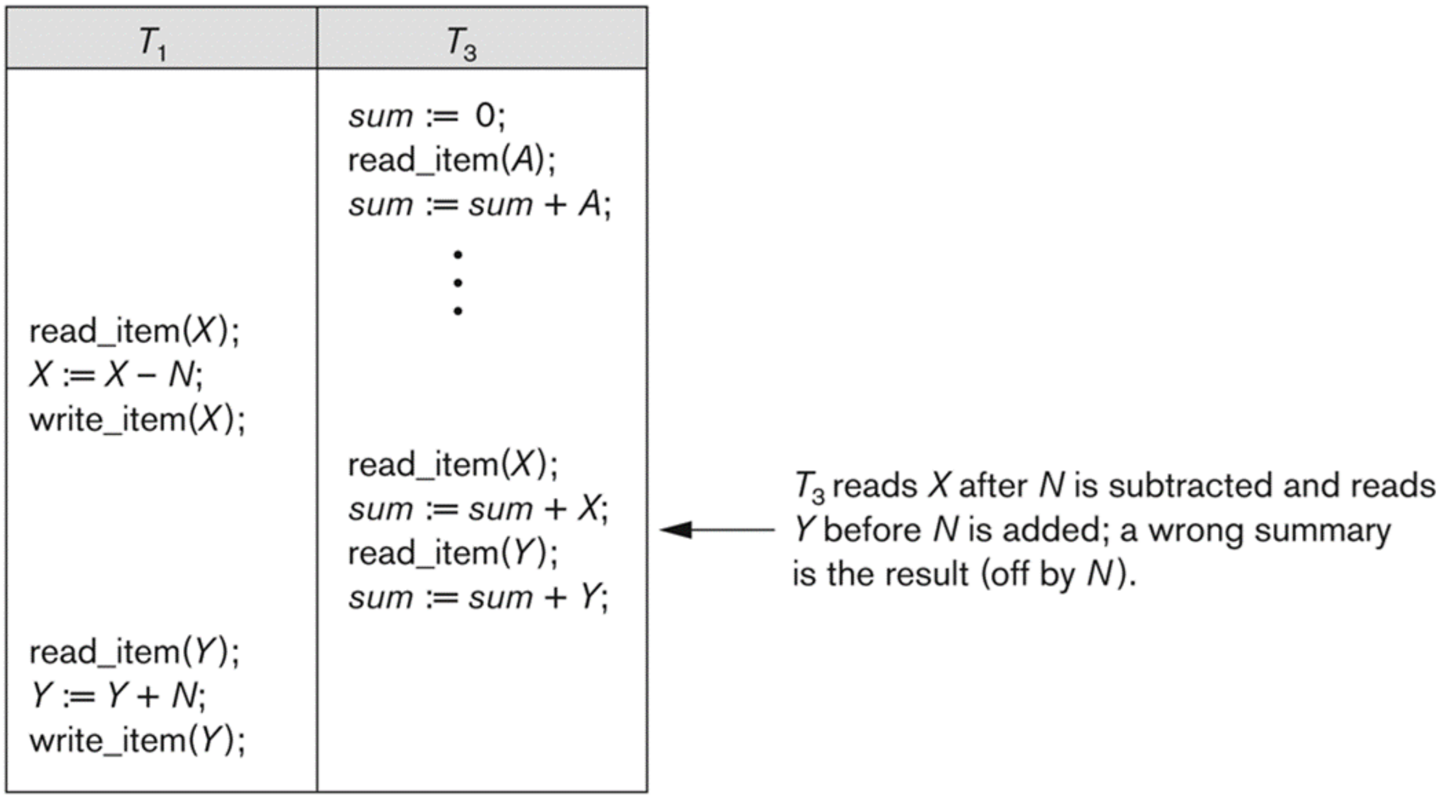
read_item(Y);	

Time
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.

Problem 3: The incorrect summary problem.

Figure 17.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

(c)

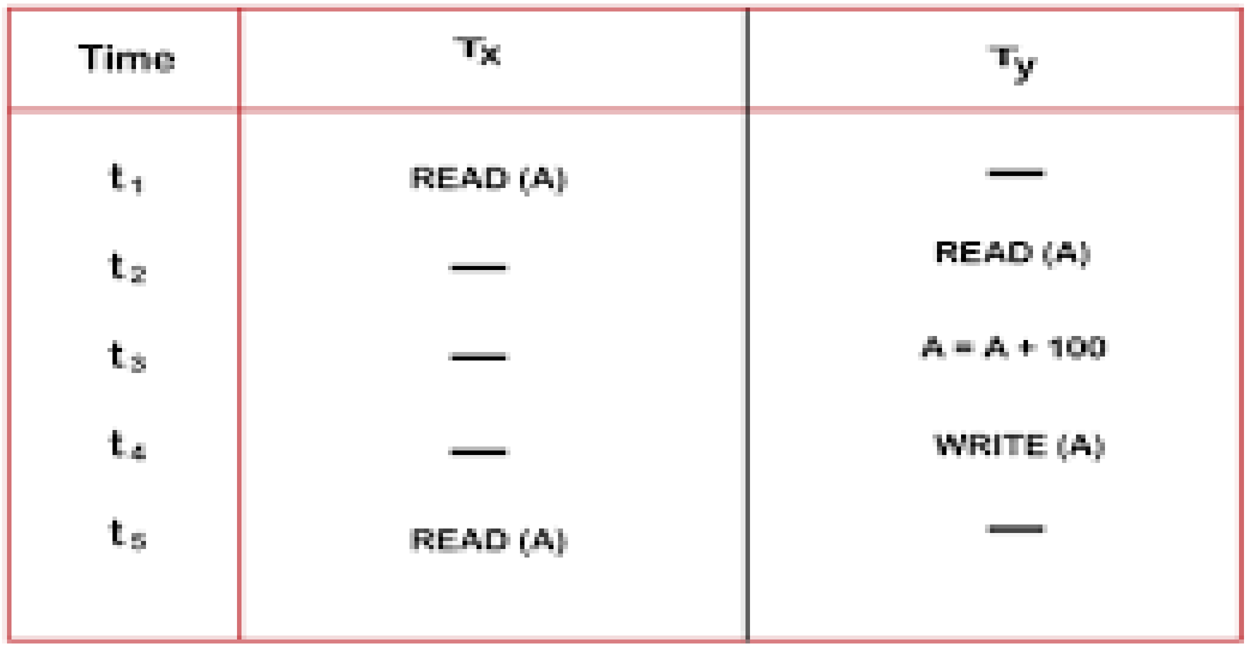


T_1	T_3
	sum := 0;
	read_item(A);
	sum := *sum* + A;
	⋮
read_item(X);	
$X := X - N$;	
write_item(X);	
	read_item(X);
	sum := *sum* + X;
	read_item(Y);
	sum := *sum* + Y;
read_item(Y);	
$Y := Y + N$;	
write_item(Y);	

T_3 reads X after N is subtracted and reads Y before N is added; a wrong summary is the result (off by N).

Problem 4: The Unrepeatable Read Problem

Consider two transactions, TX and TY, performing the read/write operations on account A, having an available balance = $300.



Time	T_x	T_y
t_1	READ (A)	—
t_2	—	READ (A)
t_3	—	A = A + 100
t_4	—	WRITE (A)
t_5	READ (A)	—

UNREPEATABLE READ PROBLEM

Serializability?

- Serializability of schedules ensures that a **non-serial schedule** is **equivalent** to a **serial schedule.** It helps in maintaining the transactions to execute simultaneously without interleaving one another.

- In simple words, serializability is a way to check if the execution of two or more transactions are maintaining the **database consistency** or not.

Types of Serializability?

1. Conflict Serializability
2. View Serializability

Conflicting Instructions

Instructions l_i and l_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both l_i and l_j, and at least one of these instructions **wrote *Q*.**

1. l_i = **read**(Q), l_j = **read**(Q). l_i and l_j don't conflict.
2. l_i = **read**(Q), l_j = **write**(Q). They conflict.
3. l_i = **write**(Q), l_j = **read**(Q). They conflict.
4. l_i = **write**(Q), l_j = **write**(Q). They conflict.

Intuitively, a conflict between l_i and l_j forces a (logical) temporal order between them.

If l_i and l_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflicting and Non-Conflicting Instructions

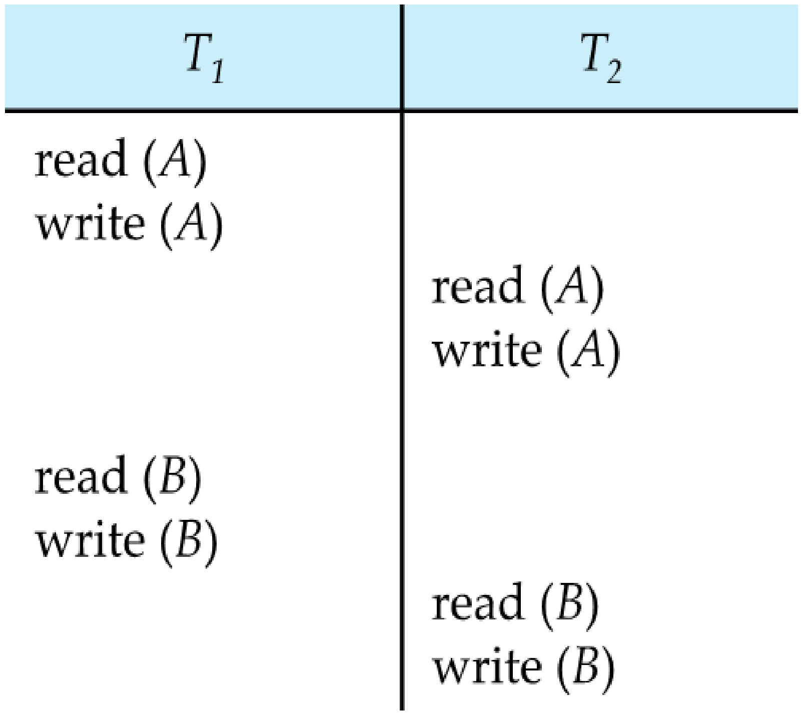
R(A)	R(A)	Non Conflict Pair
R(A)	W(A)	
W(A)	R(A)	Conflict Pairs
W(A)	W(A)	
R(B)	W(A)	
W(B)	R(A)	
R(B)	W(A)	Non-Conflict Pairs
W(A)	W(B)	

Conflict equivalent

If a schedule *S* can be transformed into a schedule *S'* by a series of swaps of non-conflicting instructions, we say that *S* and *S'* are **conflict equivalent.**

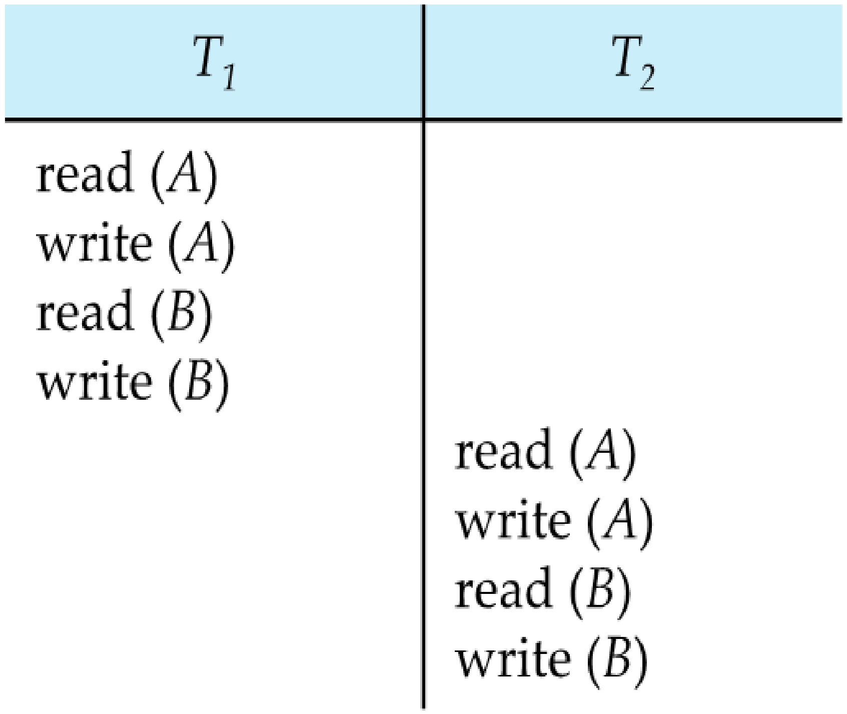
Conflict Serializability (Cont.)

Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. that is Schedule 3 and 6 is **Conflict Equivalent.** Therefore Schedule 3 is conflict serializable.



T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Schedule 3



T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Schedule 6

Conflict Serializability (Cont.)

Check whether the following schedule is conflict equivalent or not?

Question

Answer:

1. Check Conflicting pairs in other transactions and draw directed edges and draw precedence graph.
2. Check if there exist any loop/Cycle
3. If no Cycle then that schedule is **Conflict serializable.**

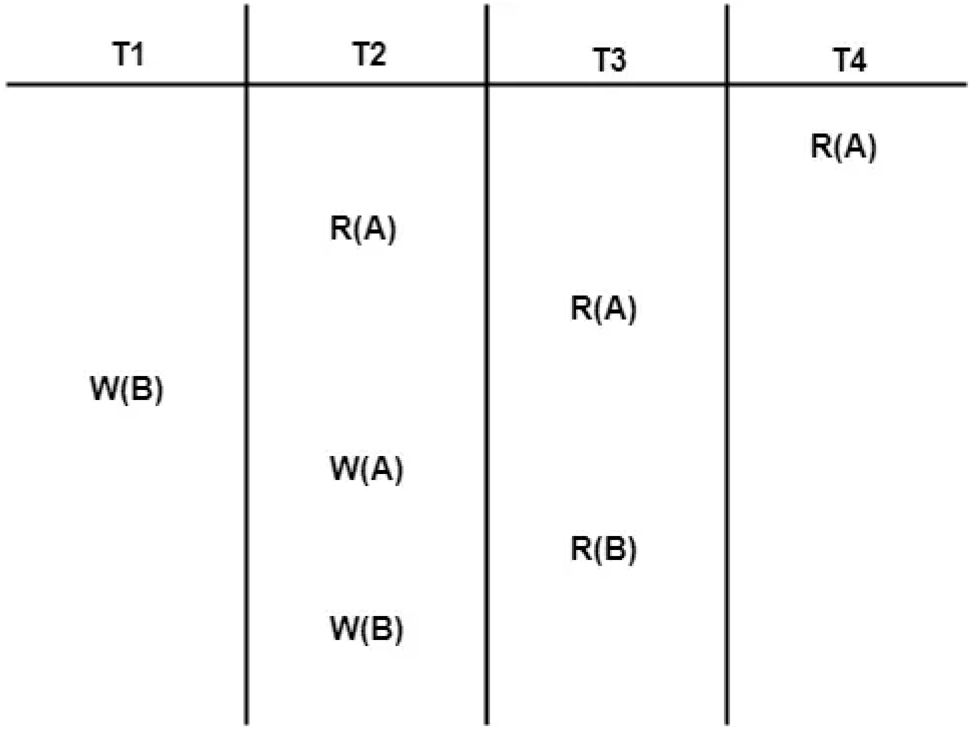
Conflict Serializability (Cont.)

Check whether the given schedule S is conflict serializable or not-

$$\mathbf{S : R_1(A) , R_2(A) , R_1(B) , R_2(B) , R_3(B) , W_1(A) , W_2(B)}$$

Conflict Serializability (Cont.)

Check whether the given schedule S is conflict serializable or not. If yes, then determine all the possible serialized schedules-



T1	T2	T3	T4
			R(A)
	R(A)		
		R(A)	
W(B)			
	W(A)		
		R(B)	
	W(B)		

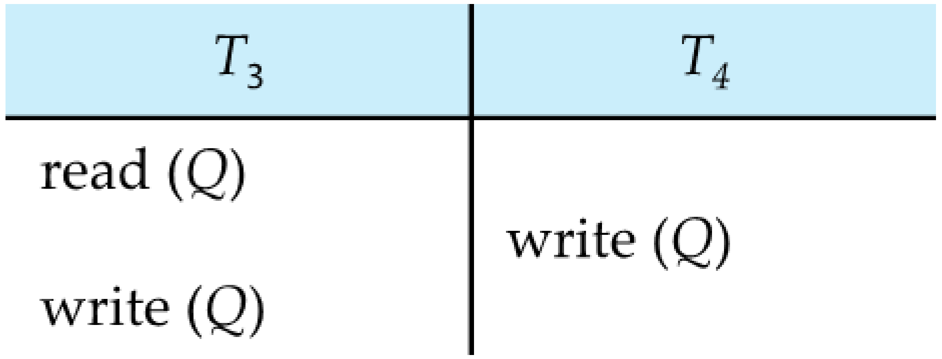
Conflict Serializability (Cont.)

Check whether the given schedule S is conflict serializable not?

T1	T2	T3	T4
	R(X)		
		W(X)	
		Commit	
W(X)			
Commit			
	W(Y)		
	R(Z)		
	Commit		
			R(X)
			R(Y)
			Commit

Conflict Serializability (Cont.)

Example of a schedule that is not conflict serializable:



T_3	T_4
read (Q)	
	write (Q)
write (Q)	

We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

View Serializability

If a given schedule is found to be **view equivalent** to some **serial schedule**, then it is called as a **view serializable schedule**.

View Serializability

Let *S* and *S'* be two schedules with the same set of transactions. *S* and *S'* are **view equivalent** if the following three conditions are met, for each data item *Q*,

1. If in schedule S, transaction T_i reads the initial value of *Q*, then in schedule *S'* also transaction T_i must read the initial value of *Q.*
2. If in schedule S transaction T_i executes **read**(*Q*), and that value was produced by transaction T_j (if any), then in schedule *S'* also transaction T_i must read the value of *Q* that was produced by the same **write**(Q) operation of transaction T_j .
3. The transaction (if any) that performs the final **write**(*Q*) operation in schedule *S* must also perform the final **write**(*Q*) operation in schedule *S'.*

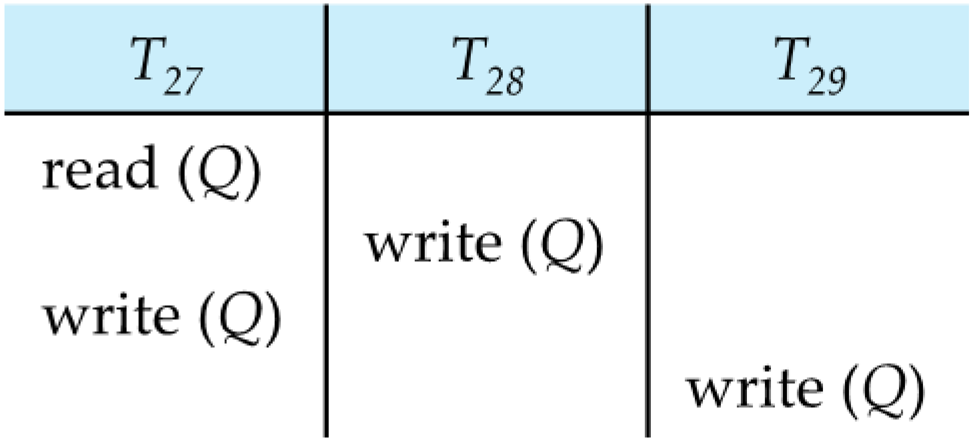
As can be seen, view equivalence is also based purely on **reads** and **writes** alone.

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 is a schedule which is view-serializable but *not* conflict serializable.



T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

What serial schedule is above equivalent to?

Every view serializable schedule that is not conflict serializable has **blind writes.**

Testing for Serializability

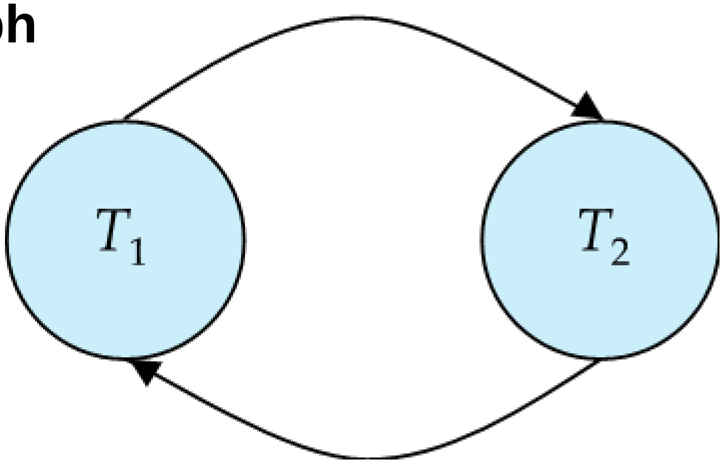
Consider some schedule of a set of transactions T_1, T_2, ..., T_n

Precedence graph — a direct graph where the vertices are the transactions (names).

We draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier.

We may label the arc by the item that was accessed.

Example of a precedence graph



Three view serializability conditions

1.Initial Read
First, we will see that transaction T1 in schedule S1 reads data item A first. Transaction T1 in S2 also reads data item A first. Then look for B. Transaction T1 in schedule S1 reads data item B first. T1 also performs the first read operation on B in S2. We checked both data items A and B, and the initial read condition is met in S1 and S2.
2.Update Read
The second method is about update read operation. Transaction T2 in S1 reads the value of A written by T1. In the same transaction, T2 reads the A after it is written by T1 in S2. Transaction T2 in S1 reads the value of B written by T1. The same transaction T2 reads the value of B after it has been updated by T1 in S2. Both schedules also satisfy the update read condition.
3.Final Write
In schedule S1, the final write operation on A is done by transaction T2. Transaction T2 also performs the final write on A in S2. Let's look for B. Transaction T2 performs the final write operation on B in schedule S1. T2 completes the final write on B in schedule S2. We checked both data items A and B, and the final write condition is met in S1 and S2.

Conditions for View Serializability:

Three Main Conditions for Checking Serializability:

“Initial readers must be same for all the data items”.
“Write-read sequence must be same.”.
“Final writers must be same for all the data items”.

•All conflict serializable schedules **are view serializable**.
•All view serializable schedules **may or may not be conflict serializable**.
•No blind write means not a **view serializable schedule**.

By using the above three conditions, write all the dependencies.
Then, draw a graph using those dependencies.
If there exists no cycle in the graph, then the schedule is view serializable otherwise not.

Note:- Blind Write – Writing without reading value of a data item.
Dirty Read - Reading same data item twice without writing anything.

Concurrency Control

- Concurrency Control is the working concept that is required for **controlling and managing the concurrent execution** of database operations and thus **avoiding the inconsistencies** in the database.
- Thus, for maintaining the consistency of the database, we have the concurrency control protocols.

Concurrency Control Protocols:

1. **Lock Based Concurrency Control Protocol**
2. **Time Stamp Concurrency Control Protocol**
3. **Validation Based Concurrency Control Protocol**

Concurrency Control Protocols

In this type of protocol, any transaction cannot read or write data until it acquires an appropriate lock on it.

There are two types of lock:

1. Shared lock:

It is also known as a **Read-only lock**. In a shared lock, the data item can only read by the transaction.

It can be shared between the transactions because when the transaction holds a lock, then it can't update the data on the data item.

Concurrency Control Protocols(cont..)

2. Exclusive lock:

- In the exclusive lock, the **data item can be both reads as well as written by the transaction**.
- This lock is exclusive, and in this lock, multiple transactions do not modify the same data simultaneously.

Concurrency Control Protocols(cont..)

There are four types of lock protocols available:

1. Simplistic lock protocol

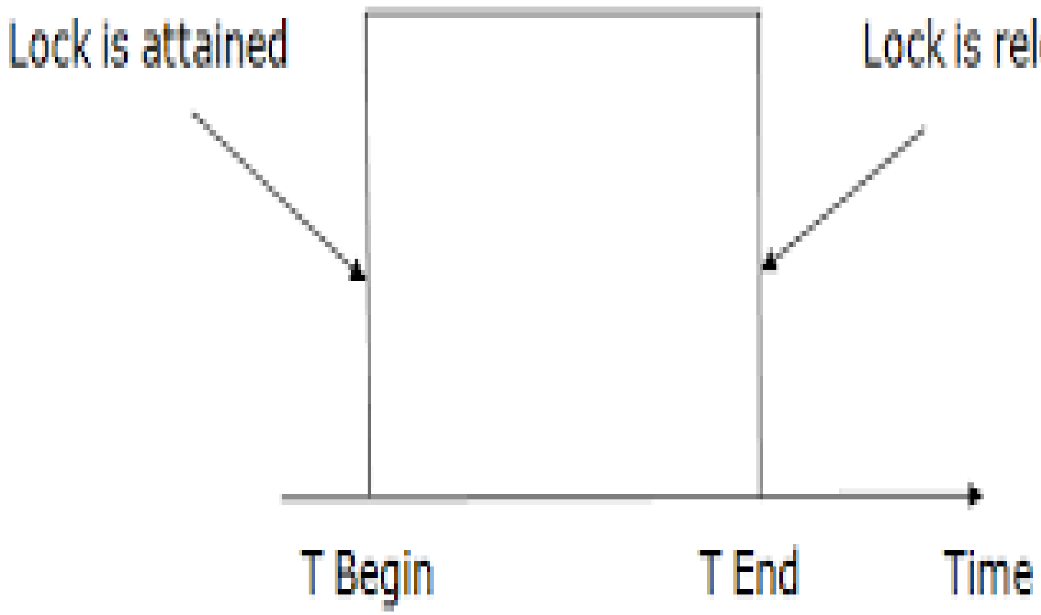
It is the simplest way of locking the data while transaction. **Simplistic lock-based protocols allow all the transactions to get the lock on the data before insert or delete or update on it.** It will unlock the data item after completing the transaction.

Concurrency Control Protocols(cont..)

2. Pre-claiming Lock Protocol

- Pre-claiming Lock Protocols evaluate the transaction to list all the data items on which they need locks.
- Before initiating an execution of the transaction, it requests DBMS for all the lock on all those data items.
- If all the locks are granted then this protocol allows the transaction to begin. When the transaction is completed then it releases all the lock.
- If all the locks are not granted then this protocol allows the transaction to rolls back and waits until all the locks are granted.

Concurrency Control Protocols(cont..)



Concurrency Control Protocols(cont..)

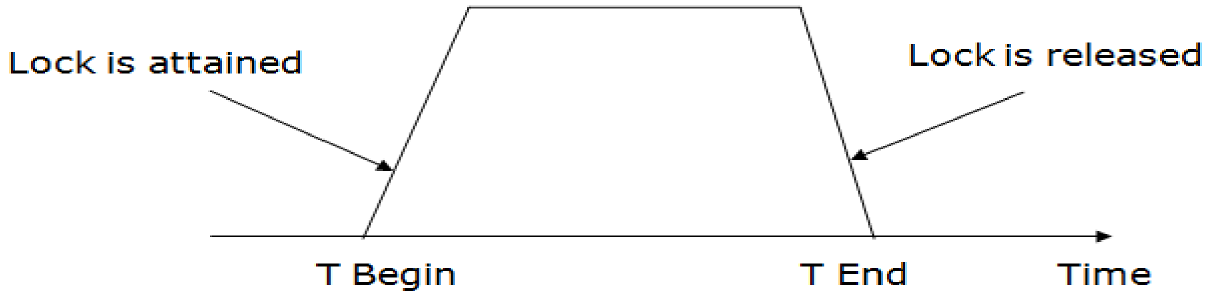
3. Two-phase locking (2PL)

The two-phase locking protocol divides the execution phase of the transaction into **three parts**.

In the **first part**, when the execution of the transaction starts, it seeks permission for the lock it requires.

In the **second part**, the transaction acquires all the locks. The third phase is started as soon as the transaction releases its first lock.

In the **third phase**, the transaction cannot demand any new locks. It only releases the acquired locks.

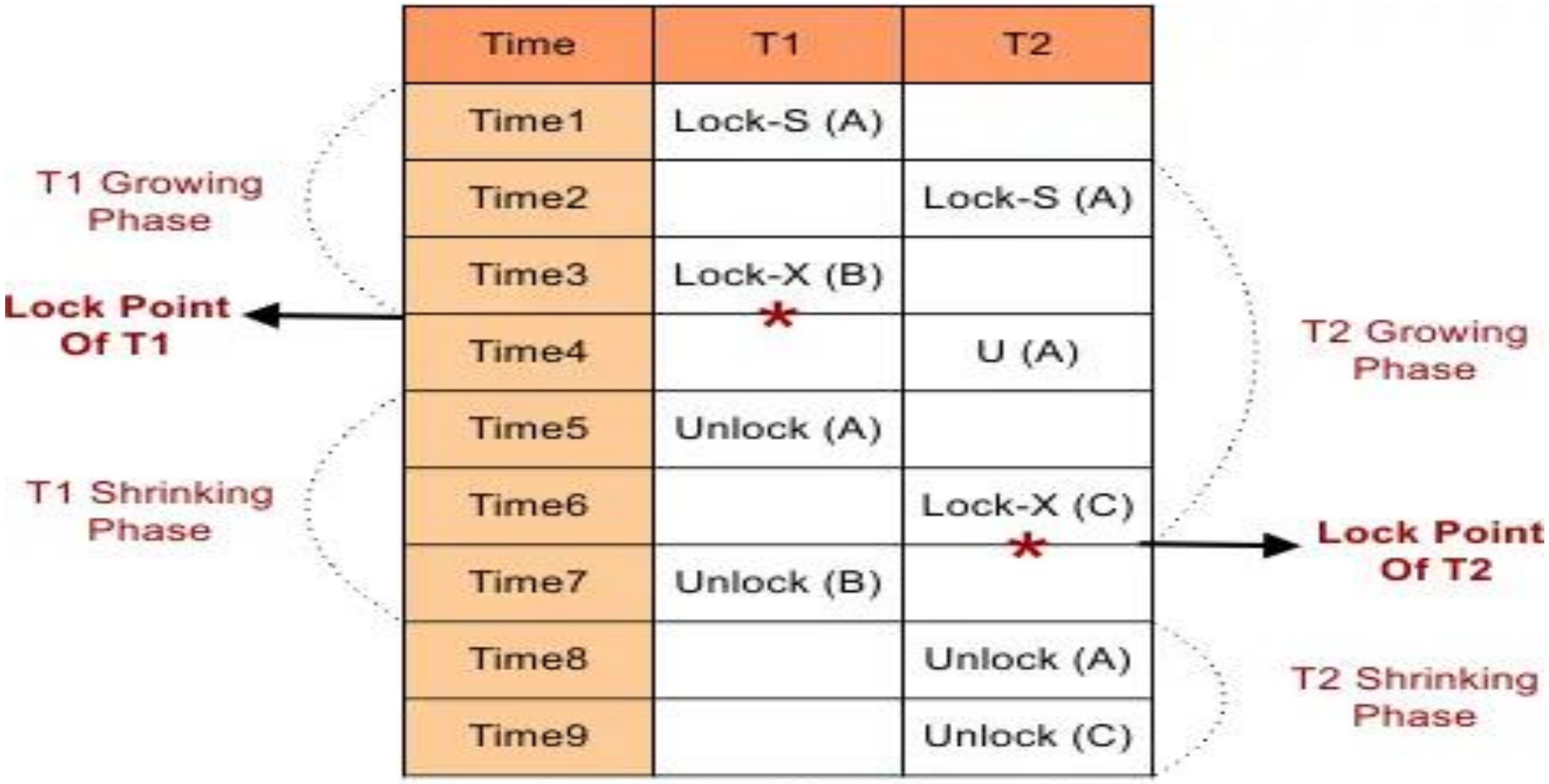


Concurrency Control Protocols(cont..)

- There are two phases of 2PL:
- **Growing phase:** In the growing phase, a new lock on the data item may be acquired by the transaction, but none can be released.
- **Shrinking phase:** In the shrinking phase, existing lock held by the transaction may be released, but no new locks can be acquired.
- In the below example, if lock conversion is allowed then the following phase can happen:
- Upgrading of lock (from S(a) to X (a)) is allowed in growing phase.
- Downgrading of lock (from X(a) to S(a)) must be done in shrinking phase.

Concurrency Control Protocols(cont..)

- **Example:**



Time	T1	T2
Time1	Lock-S (A)	
Time2		Lock-S (A)
Time3	Lock-X (B)	
Time4		U (A)
Time5	Unlock (A)	
Time6		Lock-X (C)
Time7	Unlock (B)	
Time8		Unlock (A)
Time9		Unlock (C)

T1 Growing Phase
Lock Point Of T1
T1 Shrinking Phase
T2 Growing Phase
Lock Point Of T2
T2 Shrinking Phase

2-PL Locking Example

Concurrency Control Protocols(cont..)

- **4. Strict Two-phase locking (Strict-2PL)**
- The first phase of Strict-2PL is **similar to 2PL**. In the first phase, after acquiring all the locks, the transaction continues to execute normally.
- The only difference between **2PL** and **strict 2PL** is that Strict-2PL does not release a lock after using it.
- Strict-2PL waits until the whole transaction to commit, and then it releases all the locks at a time.
- Strict-2PL protocol does not have shrinking phase of lock release.

2.Timestamp Based Protocol

- The Timestamp-based protocol is the most commonly used concurrency control protocol and is used to order the execution of the concurrent transaction based on their Timestamp. This protocol uses the **logical counter** or **system time** to determine the timestamp of the transaction. It also maintains the timestamp of last **‘read’** and **‘write’** operations on the data.
- The lock-based protocol acquires locks at the time of execution. But in this protocol, as soon as the transaction is created, it assigns the order of the concurrent transactions. The priority of an older transaction is higher; that’s why it starts its execution first.
- **Two types of Timestamp:**
- **1. W_Timestamp(X):** This represents the largest timestamp of any transaction that executes **write(X)** successfully.
- **2. R_Timestamp(X):** This represents the largest timestamp of any transaction that executes **Read(X)** successfully.

Timestamp ordering protocol

- The timestamp ordering protocol ensures that any conflicting read and write operation are executed in timestamp order.
- **Following are the two cases which describe the work of basic timestamp ordering protocol:**
- 1. Suppose a transaction Ti read an item (X), check the following condition:
- If Timestamp(Ti) < W_Timestamp(X), then the read operation is rejected and Ti is rolled back.
- If Timestamp(Ti) > W_Timestamp(X), then the operation is executed.
- 2. Suppose a transaction Ti write an item (X), check the following condition:
- If Timestamp(Ti) < R_Timestamp(X) then the operation is rejected.
- If Timestamp(Ti) < W_Timestamp(X) then the write operation is rejected and Ti transaction is rolled back otherwise, other operations are executed.

Recoverability of Schedule

1.Log-Based Recovery

•The log is a **sequence of records**. Log of each transaction is maintained in some stable storage so that if any failure occurs, then it can be recovered from there.

•If any operation is performed on the database, then it will be recorded in the log.

•But the process of storing the logs should be done before the actual transaction is applied in the database.

Let's assume there is a transaction to modify the City of a student. The following logs are written for this transaction.

•When the transaction is initiated, then it writes 'start' log.

<Tn, Start>

•When the transaction modifies the City from 'Noida' to 'Bangalore', then another log is written to the file.

<Tn, City, 'Noida', 'Bangalore' >

•When the transaction is finished, then it writes another log to indicate the end of the transaction.

<Tn, Commit>

Recoverability of Schedule

There are **two approaches** to modify the database:

1. Deferred database modification:

•The deferred modification technique occurs if the transaction does not modify the database until it has committed.

•In this method, all the logs are created and stored in the stable storage, and the database is updated when a **transaction commits**.

2. Immediate database modification:

•The Immediate modification technique occurs if database modification occurs while the transaction is still active.

•In this technique, the database is modified immediately after every operation. It follows an actual database modification.

Recoverability of Schedule

Recovery using Log records

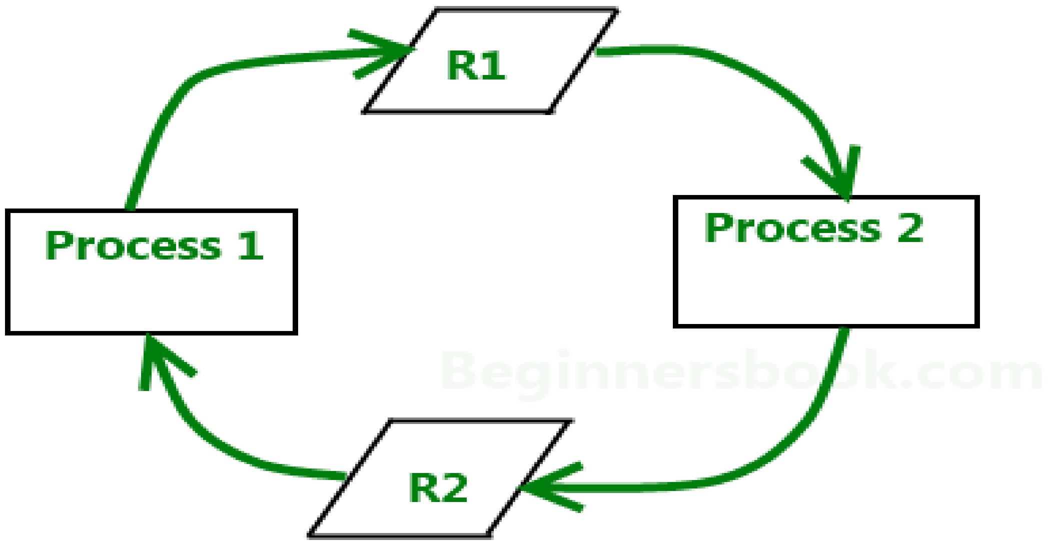
When the system is **crashed**, then the system consults the log to find which transactions need to be undone and which need to be redone.

1.If the log contains the record <Ti, Start> and <Ti, Commit> or <Ti, Commit>, then the Transaction Ti needs to be redone.

2.If log contains record<T_n, Start> but does not contain the record either <Ti, commit> or <Ti, abort>, then the Transaction Ti needs to be undone.

Deadlock in DBMS

- A deadlock is a condition wherein two or more tasks are waiting for each other in order to be finished but none of the task is willing to give up the resources that other task needs. In this situation no task ever gets finished and is in waiting state forever



Process P1 holds resource R2 and requires R1 while
Process P2 holds resource R1 and requires R2.

Examples of Deadlock in DBMS

For example: In the student table, transaction T1 holds a lock on some rows and needs to update some rows in the grade table. Simultaneously, transaction T2 holds locks on some rows in the grade table and needs to update the rows in the Student table held by Transaction T1.

Now, the main problem arises. Now Transaction T1 is waiting for T2 to release its lock and similarly, transaction T2 is waiting for T1 to release its lock. All activities come to a halt state and remain at a standstill. It will remain in a standstill until the DBMS detects the deadlock and aborts one of the transactions

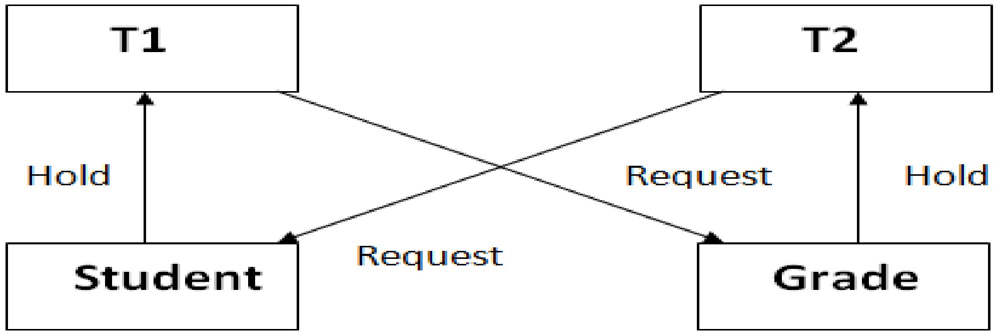
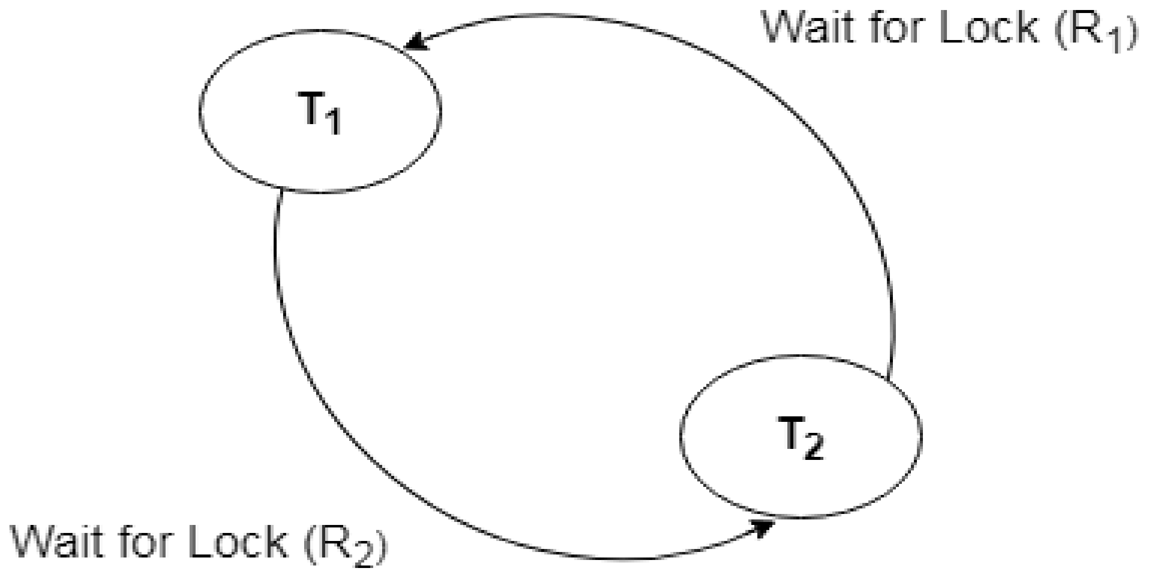


Figure: Deadlock in DBMS

Deadlock Detection



Deadlock Detection

- In a database, when a transaction waits indefinitely to obtain a lock, then the DBMS should detect whether the transaction is involved in a deadlock or not. The lock manager maintains a Wait for the graph to detect the deadlock cycle in the database.
- **Wait for Graph**
- This is the suitable method for **deadlock detection.** In this method, a **graph** is created based on the t**ransaction and their lock**. If the created graph has a cycle or closed loop, then there is a deadlock.
- The wait for the graph is maintained by the system for every transaction which is waiting for some data held by the others. The system keeps checking the graph if there is any cycle in the graph.

Deadlock Detection

1.Wait-Die scheme

In this scheme, if a transaction requests for a resource which is already held with a conflicting lock by another transaction then the DBMS simply checks the timestamp of both transactions. It allows the older transaction to wait until the resource is available for execution.

- Consider there are two transactions Ti and Tj and let TS(T) is a timestamp of any transaction T. If T2 holds a lock by some other transaction and T1 is requesting for resources held by T2 then the following actions are performed by DBMS:

- Check if TS(Ti) < TS(Tj) - If Ti is the older transaction and Tj has held some resource, then Ti is allowed to wait until the data-item is available for execution. That means if the older transaction is waiting for a resource which is locked by the younger transaction, then the older transaction is allowed to wait for resource until it is available.

- Check if TS(T_i) < TS(Tj) - If Ti is older transaction and has held some resource and if Tj is waiting for it, then Tj is killed and restarted later with the random delay but with the same timestamp.

Deadlock Detection

2.Wound wait scheme

- In wound wait scheme, if the older transaction requests for a resource which is held by the younger transaction, then older transaction forces younger one to kill the transaction and release the resource. After the minute delay, the younger transaction is restarted but with the same timestamp.
- If the older transaction has held a resource which is requested by the Younger transaction, then the younger transaction is asked to wait until older releases it.