UNIT IV

Database Transaction Management

Introduction- Transaction

- Collections of operations that form a single logical unit of work are called transactions.

OR

- A ***transaction*** is a *unit* of program execution that accesses and possibly updates various data items.
- At the end of transaction database must be in consistence state.
- A transaction is delimited by statements (or function calls) of the form **begin transaction** and **end transaction.**
- The transaction consists of all operations executed between the **begin transaction** and **end transaction.**

Introduction- Transaction

- **Two main issues to deal with:**
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

- **Transactions access data using two operations:**
  - **read(X)**, which transfers the data item X from the database to a variable, also called X, in a buffer in main memory belonging to the transaction that executed the read operation.
  - **write(X)**, which transfers the value in the variable X in the main-memory buffer of the transaction that executed the write to the data item X in the database.

ACID Properties

To preserve integrity of data, the database system must ensure:

- **Atomicity:** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency:** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation:** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
  - That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.
- **Durability:** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Example of Fund Transfer

- Transaction to transfer $50 from account *A* to account *B*:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)

- **Consistency requirement** – the sum of *A* and *B* is unchanged by the execution of the transaction.

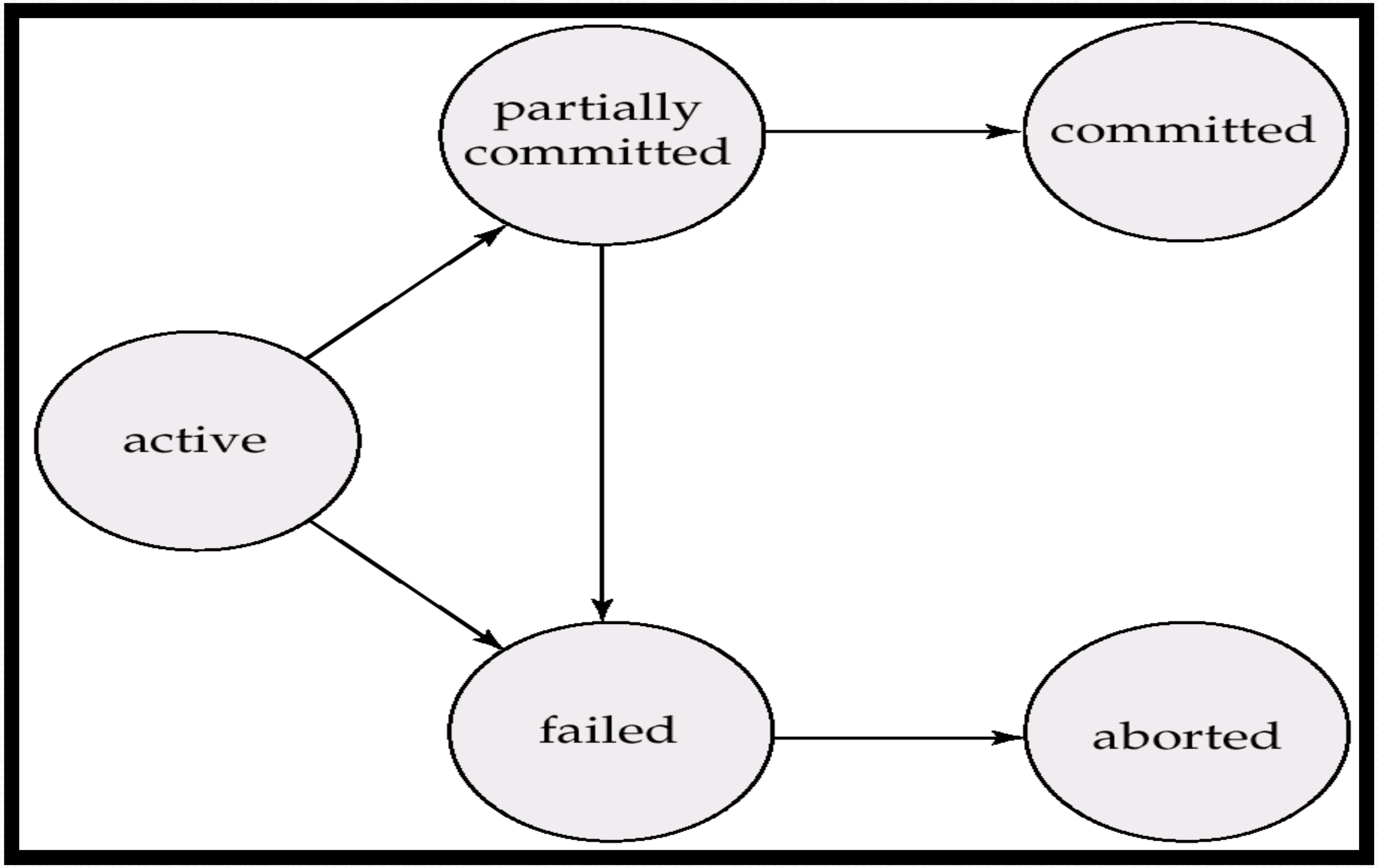
- **Atomicity requirement** — if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.

Example of Fund Transfer

- **Durability requirement —** once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist despite failures.

- **Isolation requirement —** if between steps 3 and 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be). Can be ensured trivially by running transactions ***serially***, that is one after the other. However, executing multiple transactions concurrently has significant benefits, as we will see.

Transaction State



Transaction State (Contd…)

- **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.

- **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 transaction aborted due to h/w or s/w error.
  - **kill the transaction** – logical error
- **Committed-** after successful completion.

Concurrent Executions

- **Multiple transactions** are allowed to run concurrently in the system.
- 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- *The number of transactions executed in a given amount of time.*

  - **Reduced average response time** for transactions: short transactions need not wait behind long ones.

  Response time- *The average time for a transaction to be completed after it has been submitted.*

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

Schedules

- **Schedule –** 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.
- A transaction that **successfully** completes its execution will have a **commit instructions** as the last statement.
  - 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.

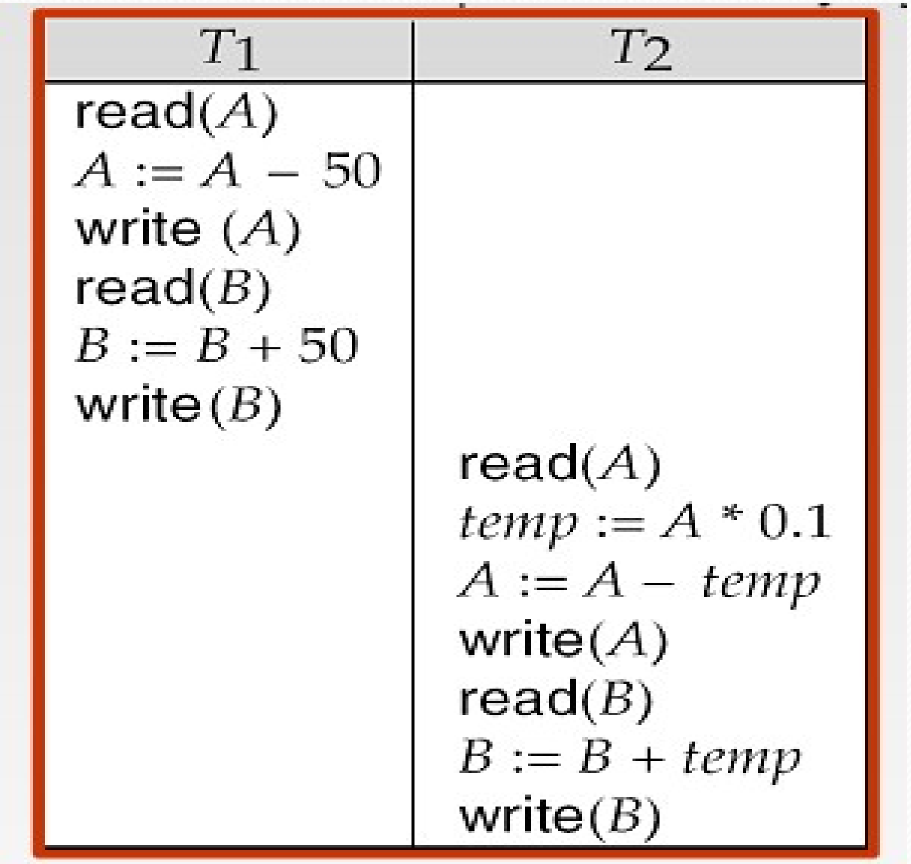
Types of Schedules

- Serial Schedule – It consist of a sequence of instructions from various transactions, where the instructions belonging to one single transaction appear together in that schedule.

- Concurrent Schedule - If the operating system is executing one transaction for a little while, then perform a context switch, execute the second transaction for some time, and then switch back to the first transaction for some time, and so on.

Example: Schedules-1

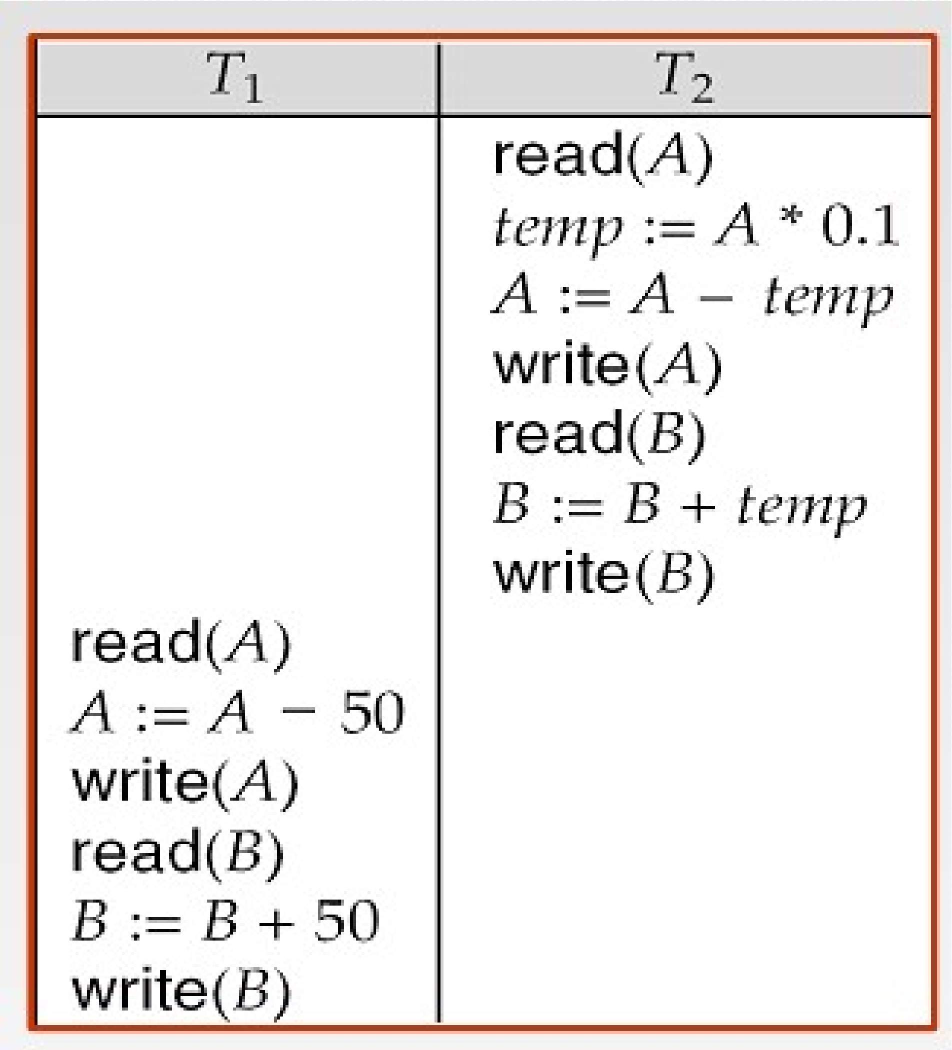
- Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B. Initially $A = 1000$ and $B = 2000$
- ***A Serial schedule*** in which T_1 is followed by T_2 :



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

Example: Schedules-2

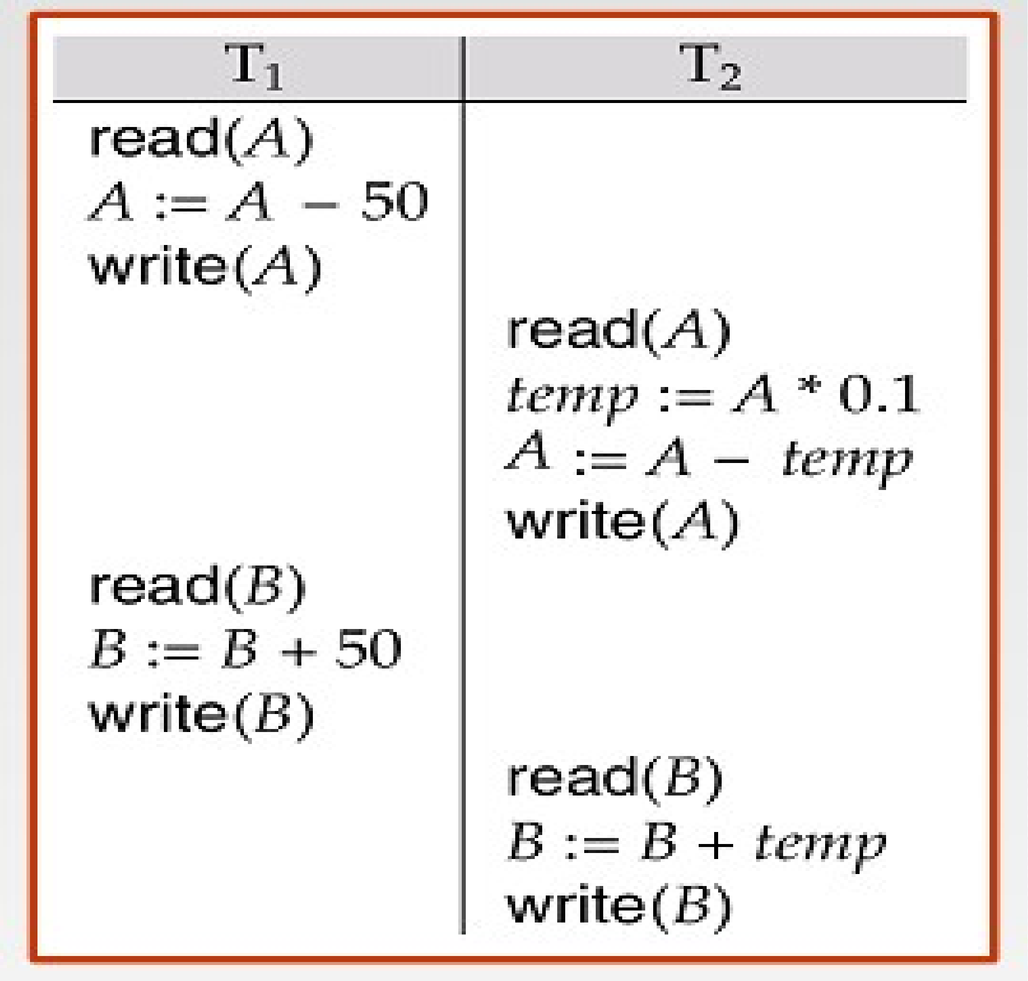
- ***A Serial schedule*** in which T_2 is followed by T_1 :



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

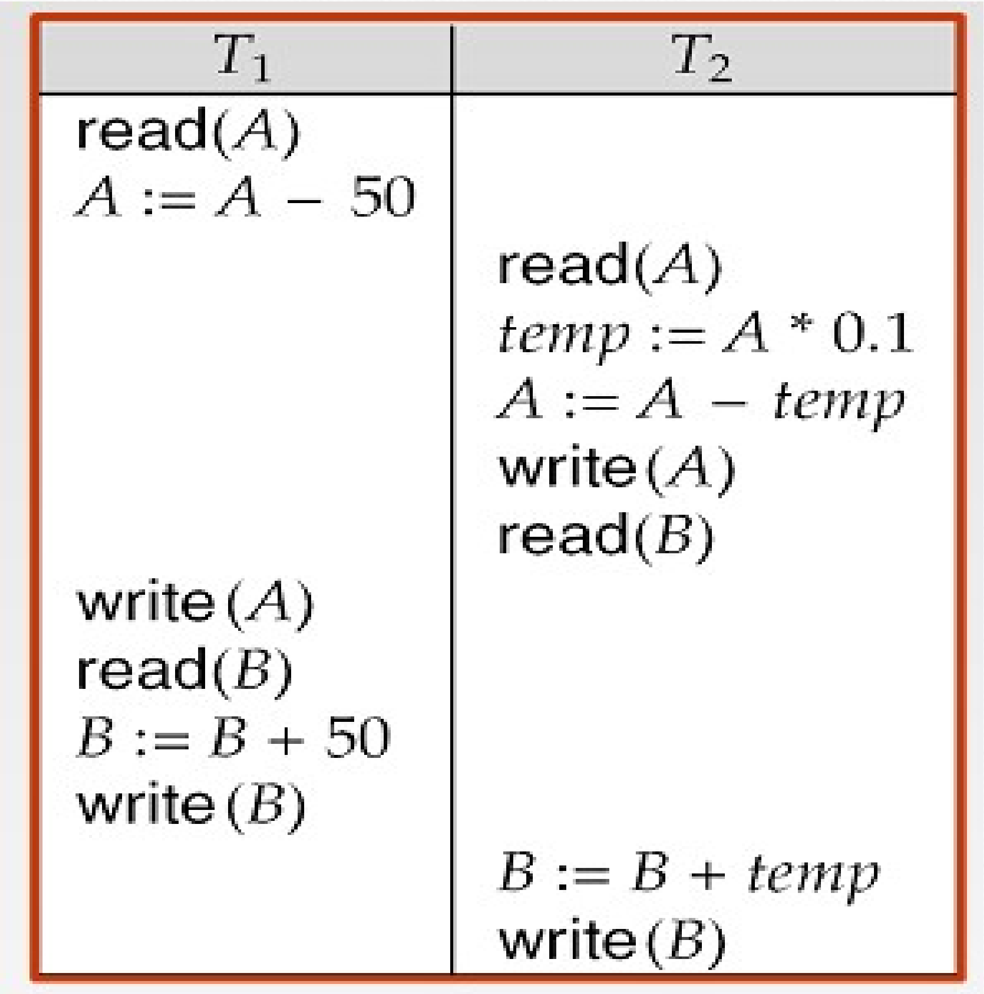
Example: Schedules-3

- Let T_1 and T_2 be the transactions defined previously.
- The following schedule is ***a concurrent schedule***.



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)	
	read(B) $B := B + temp$ write(B)

Example: Schedules-4



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)	
	$B := B + temp$
	write(B)

Concurrent Executions Contd…

- If control of concurrent execution is left entirely to the operating system, many possible schedules, including ones that leave the database in an inconsistent state, such as the one just described, are possible.
- It is the job of the database system to ensure that any schedule that is executed will leave the database in a consistent state.
- The **concurrency-control component** of the database system carries out this task.
- The method used to check consistency of database is called **serializability.**

Serializability

- **Serializability** is a method to find wheather the given schedule is serializable or not.
- All serializable schedules preserves database consistency.
- Serial execution of a set of transactions preserves database consistency, so all **serial schedules** are serializable.
- Not all **concurrent schedule** is serializable, but it can be serializable if it is equivalent to a serial schedule.

➢ **Conflict serializability**

➢ **View serializability**

Conflict Serializability

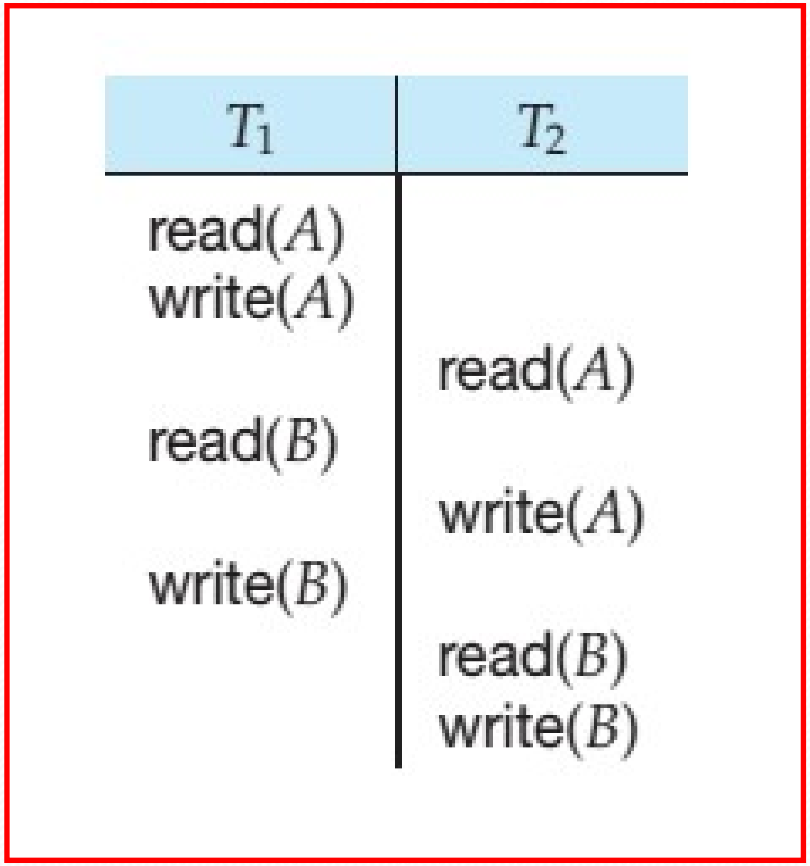
- Consider a schedule S in which there are two consecutive instructions Li and Lj of transactions Ti and Tj respectively

- If Li and Lj refer to different data items, that are **non conflict** instructions and if we can swap such instructions without affecting result.

- Instructions **conflict** if and only if there exists same data item Q accessed by both Li and Lj.

- Li = read(Q), Lj = read(Q). Li and Lj don't conflict.
- Li = read(Q), Lj = write(Q). They conflict.
- Li = write(Q), Lj = read(Q). They conflict
- Li = write(Q), Lj = write(Q). They conflict

Conflict Serializability



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

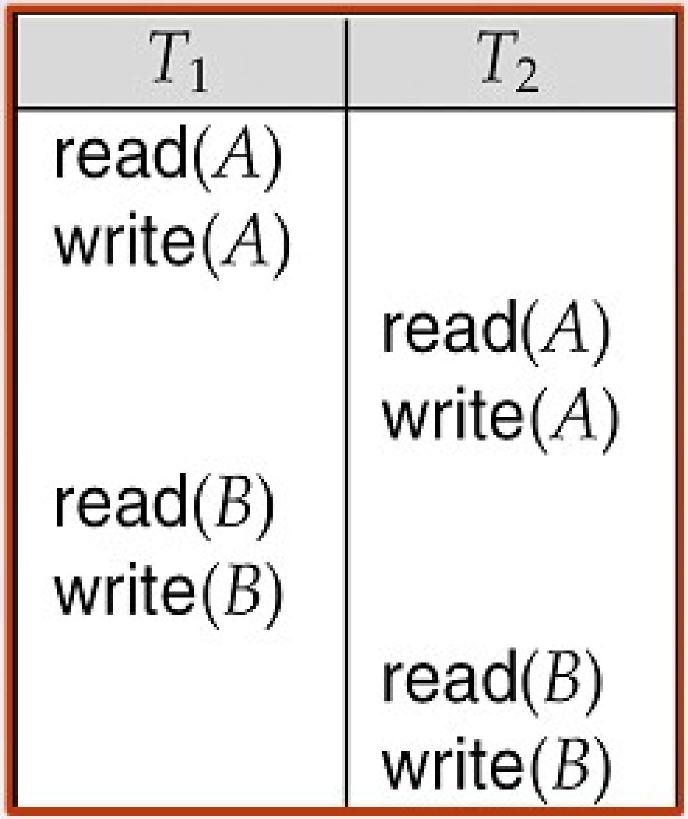
Schedule 5

Conflict Serializability (Contd…)

- 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.
- We say that a schedule S is conflict serializable if it is conflict equivalent to a serial schedule.

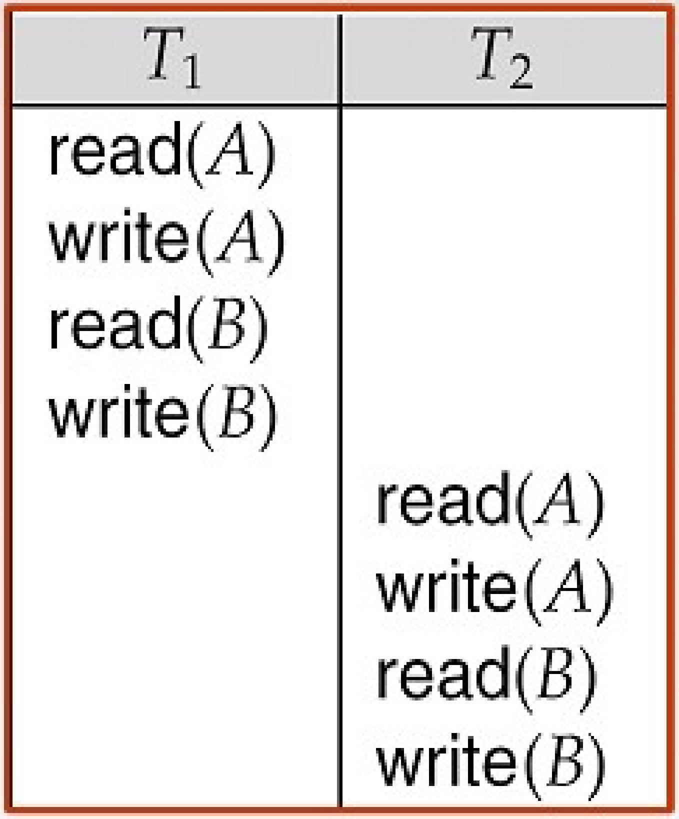
Conflict Serializability (Contd…)

Therefore schedule 3 is conflict serializable, since it is conflict equivalent to the serial schedule 1.



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

Schedule 3

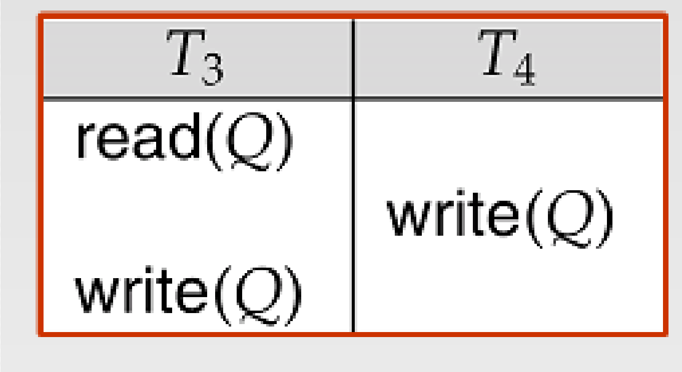


T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Schedule 6

Conflict Serializability (Contd…)

- Example of a schedule-7



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>.

- Schedule is not conflict serializable

View Serializability

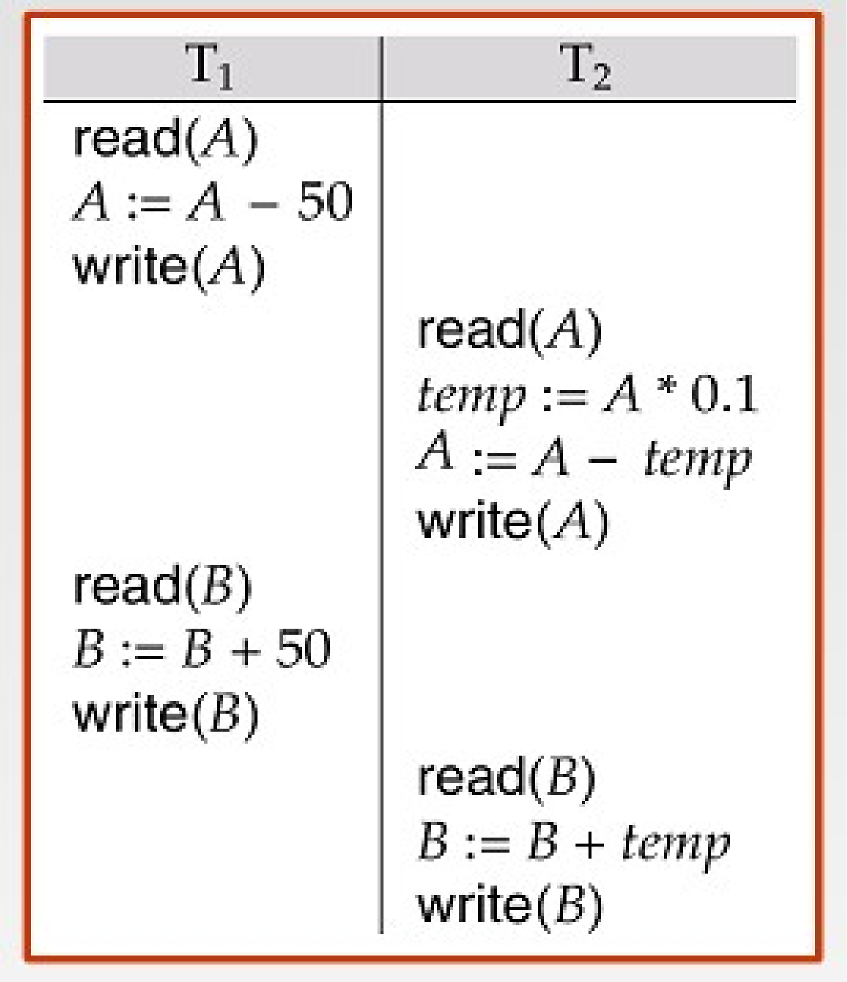
- Let S and S´ be two schedules with the same set of transactions. S and S´ are view equivalent if the following three conditions are met, for each data item Q:

- If in schedule S, transaction Ti reads the **initial value** of Q, then in schedule S' also transaction Ti must read the **initial value** of Q.

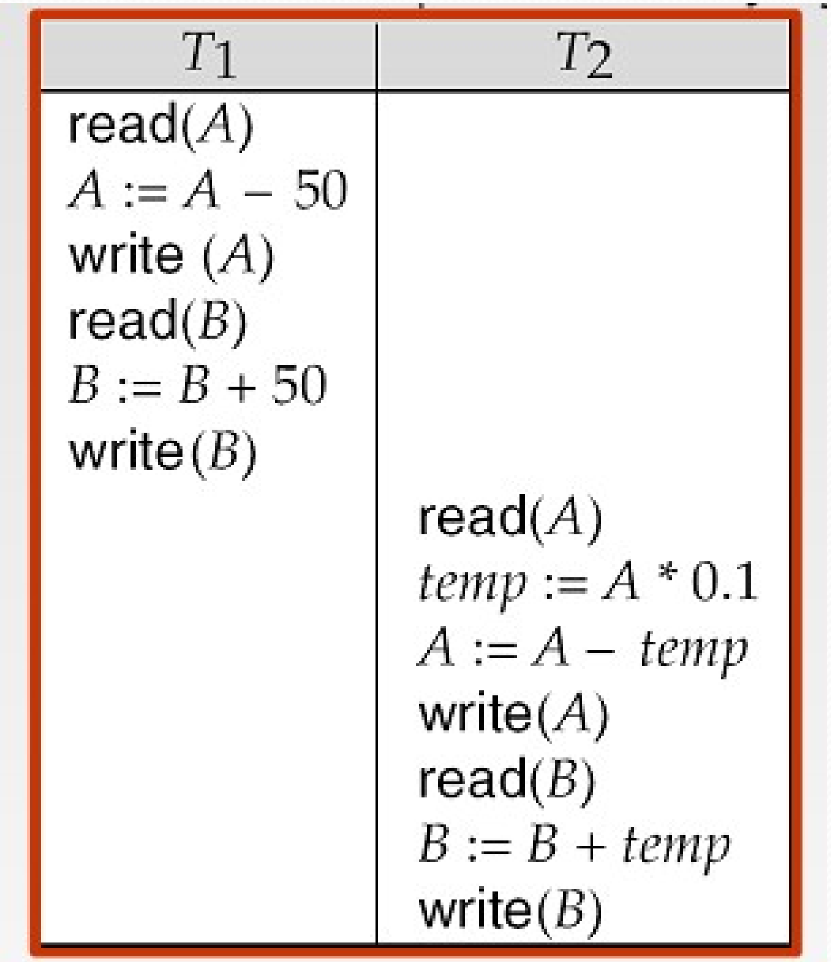
- If in schedule S transaction Ti executes **read(Q)**, and that value was produced by transaction Tj (if any), then in schedule S' also transaction Ti must read the value of Q that was produced by the same **write(Q)** operation of transaction Tj.

- 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'.

View Serializability



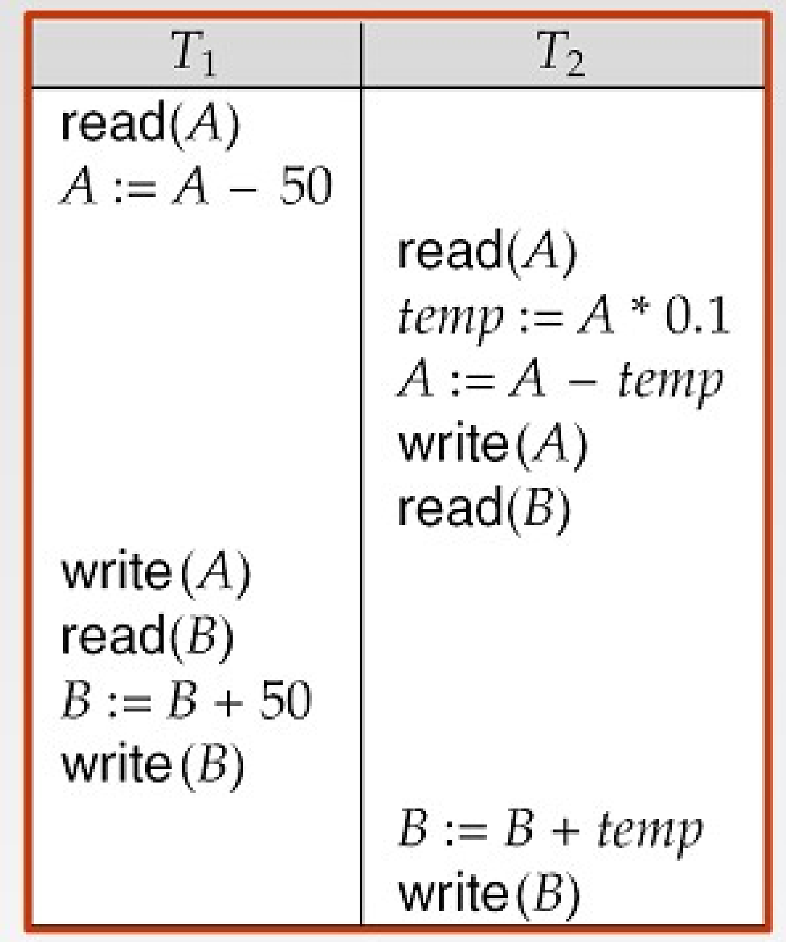
T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)



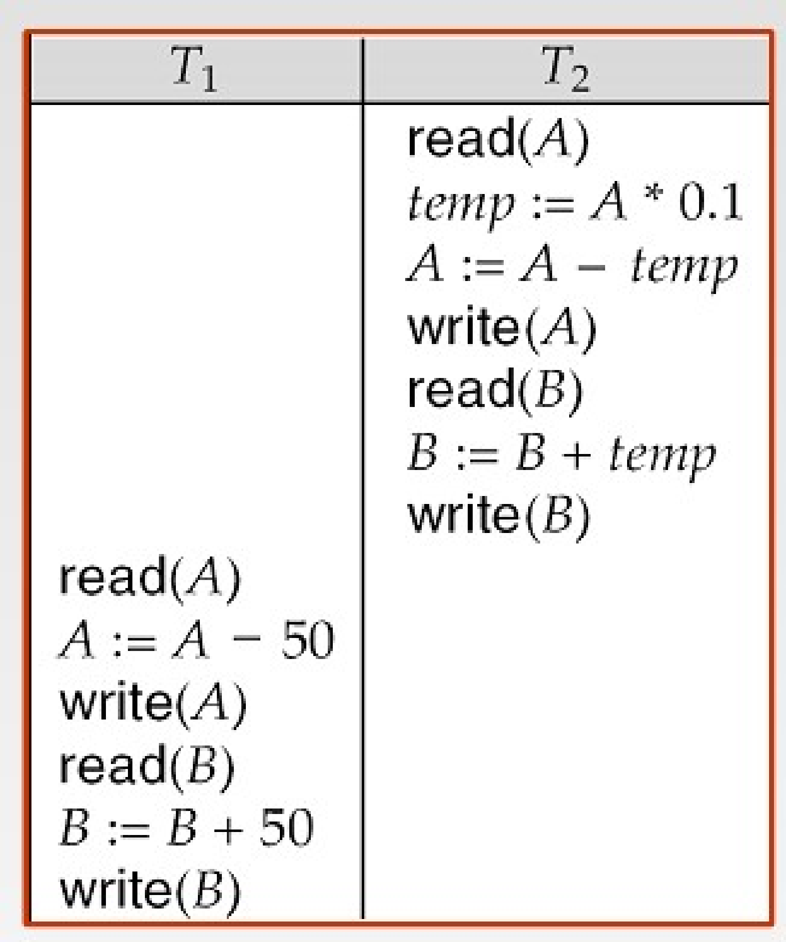
$T1$	$T2$
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

A schedule S is view serializable if it is view equivalent to a serial schedule.

View Serializability

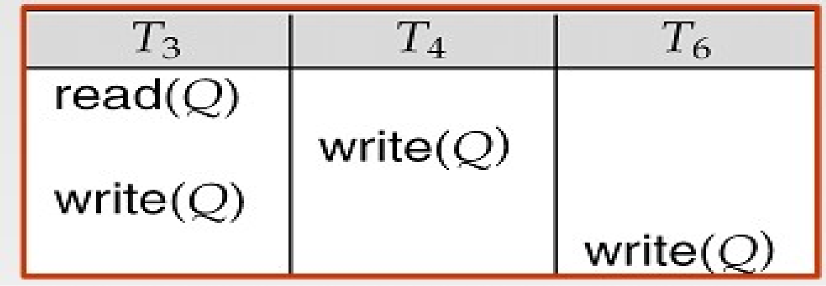


T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)



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

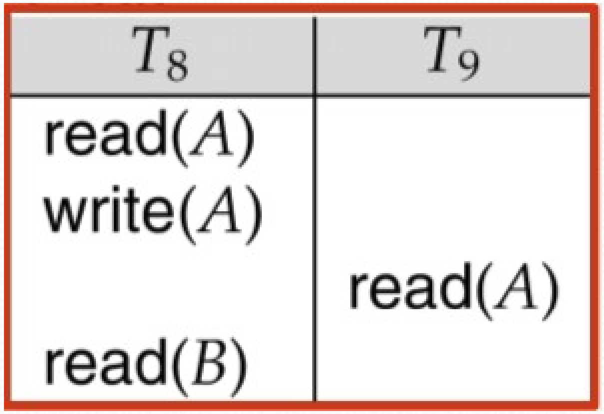
View Serializability



T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

- Transactions T_4 and T_6 perform write(Q) operations without having performed a read(Q) operation.
- Writes of this sort are called **blind writes**.
- Blind writes appear in any view-serializable schedule that is not conflict serializable.

Non Recoverable Schedule



T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

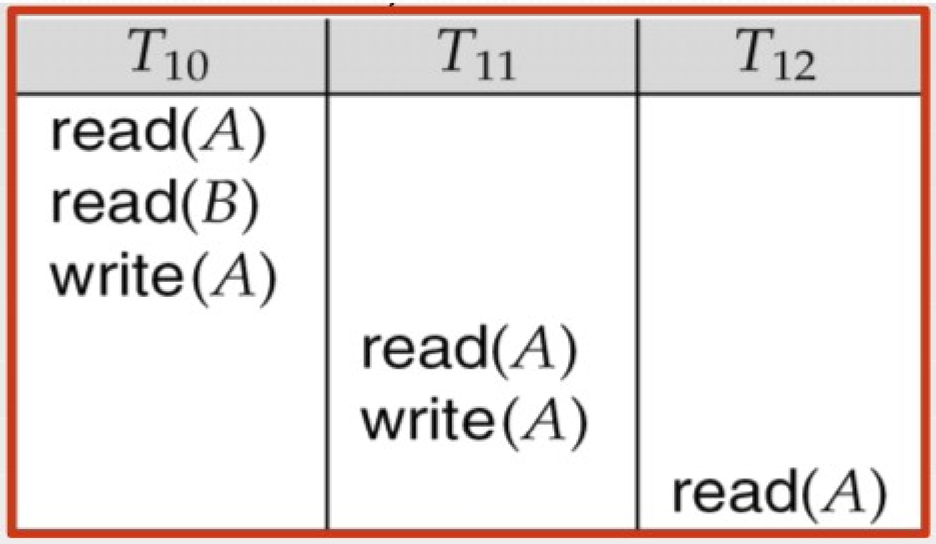
The above schedule is not recoverable if T9 commits immediately after the read

Recoverable schedule — If a transaction Tj reads a data item previously written by a transaction Ti, then the commit operation of Ti appears before the commit operation of Tj .

Cascading aborts

Cascading aborts/rollback – a single transaction failure leads to a series of transaction rollbacks.

Consider the following schedule where none of the transactions has yet committed



T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

If T10 fails, T11 and T12 must also be rolled back.

Can lead to the undoing of a significant amount of work

Cascadeless schedules

- **Cascadeless schedules** — cascading aborts/rollbacks cannot occur; for each pair of transactions Ti and Tj such that Tj reads a data item previously written by Ti, the commit operation of Ti appears before the read operation of Tj .

Part-B

Concurrency Control

Concurrency Control

- The system must control the interaction among the concurrent transactions; this control is achieved through mechanisms called **concurrency control**.

- Type of Protocol used for **concurrency control:**

➢ Lock-Based Protocols

➢ Timestamp-Based Protocols

Lock-Based Protocols

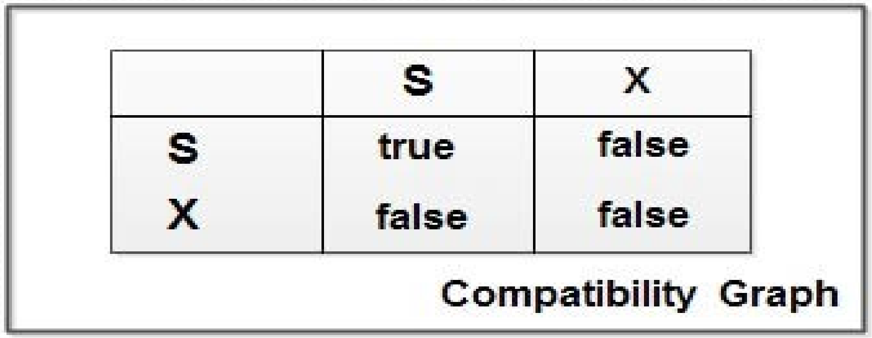
- One way to ensure serializability is to require that data items be accessed in a mutually exclusive manner.
- A **lock** is a mechanism to control concurrent access to a data item.
- Data items can be locked in two modes :

- **Shared (S) mode**-Data item can only read and cannot write. S-lock is requested using lock-S instruction.

- **Exclusive (X) mode**-Data item can be both read as well as written. X-lock is requested using lock-X instruction.

Lock Based Protocols

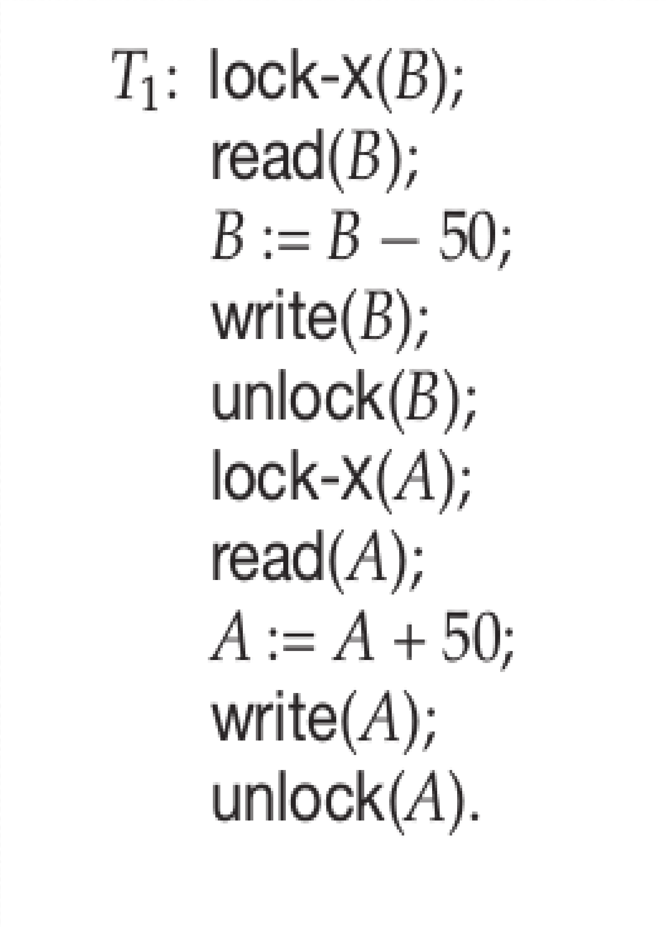
- Transaction request for a appropriate lock depending on the types of operations that it will perform on data item.
- Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.
- Compatibility Function



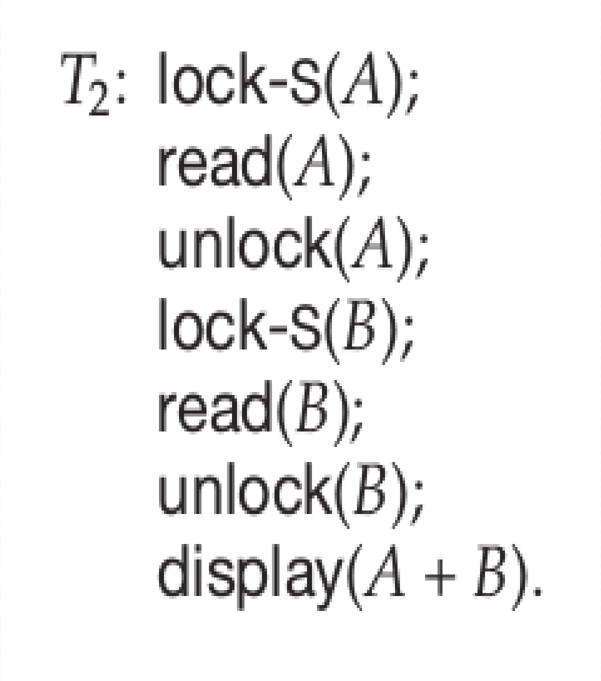
	S	X
S	true	false
X	false	false

Compatibility Graph

Granting of Locks

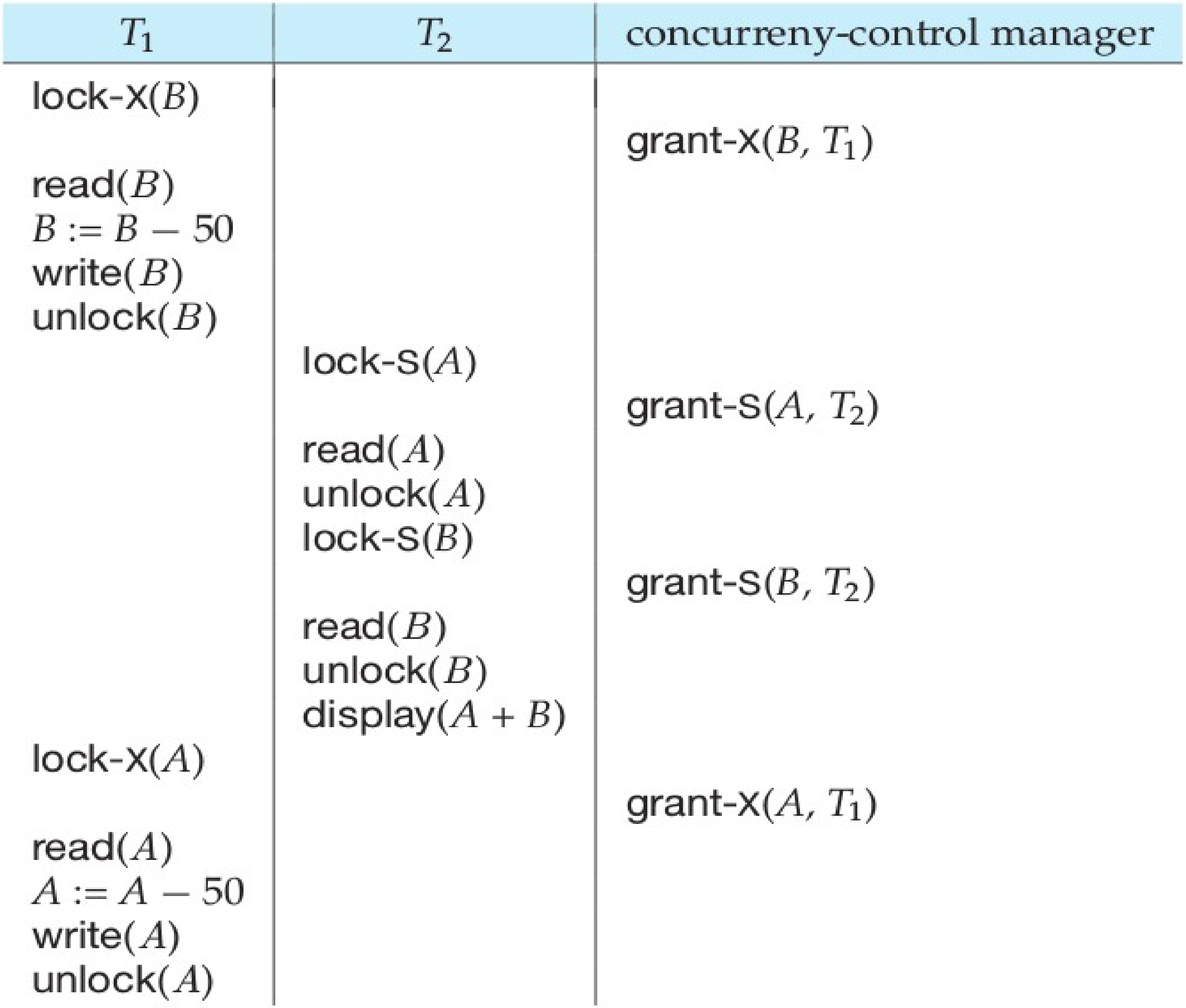


```
$T_1$: lock-X($B$);
       read($B$);
       $B := B - 50$;
       write($B$);
       unlock($B$);
       lock-X($A$);
       read($A$);
       $A := A + 50$;
       write($A$);
       unlock($A$).
```



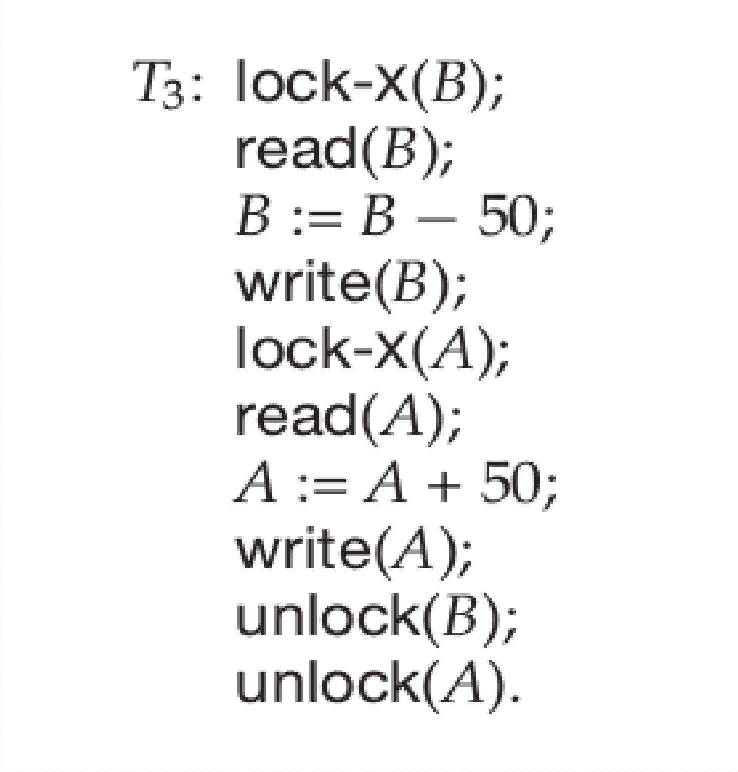
```
$T_2$: lock-S($A$);
       read($A$);
       unlock($A$);
       lock-S($B$);
       read($B$);
       unlock($B$);
       display($A + B$).
```

Granting of Locks

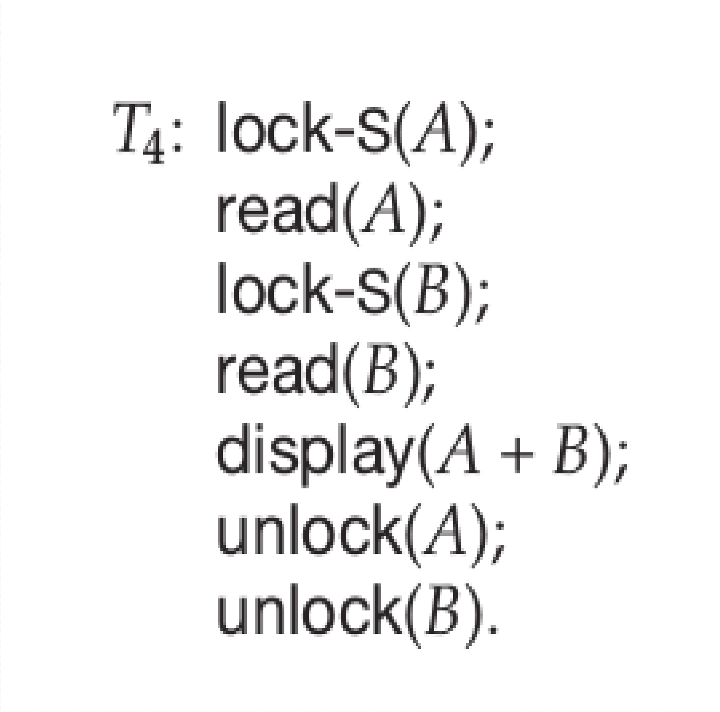


T_1	T_2	concurreny-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B) $B := B - 50$ write(B) unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A) unlock(A) lock-S(B)	
		grant-S(B, T_2)
	read(B) unlock(B) display($A + B$)	
lock-X(A)		
		grant-X(A, T_1)
read(A) $A := A - 50$ write(A) unlock(A)		

Granting of Locks

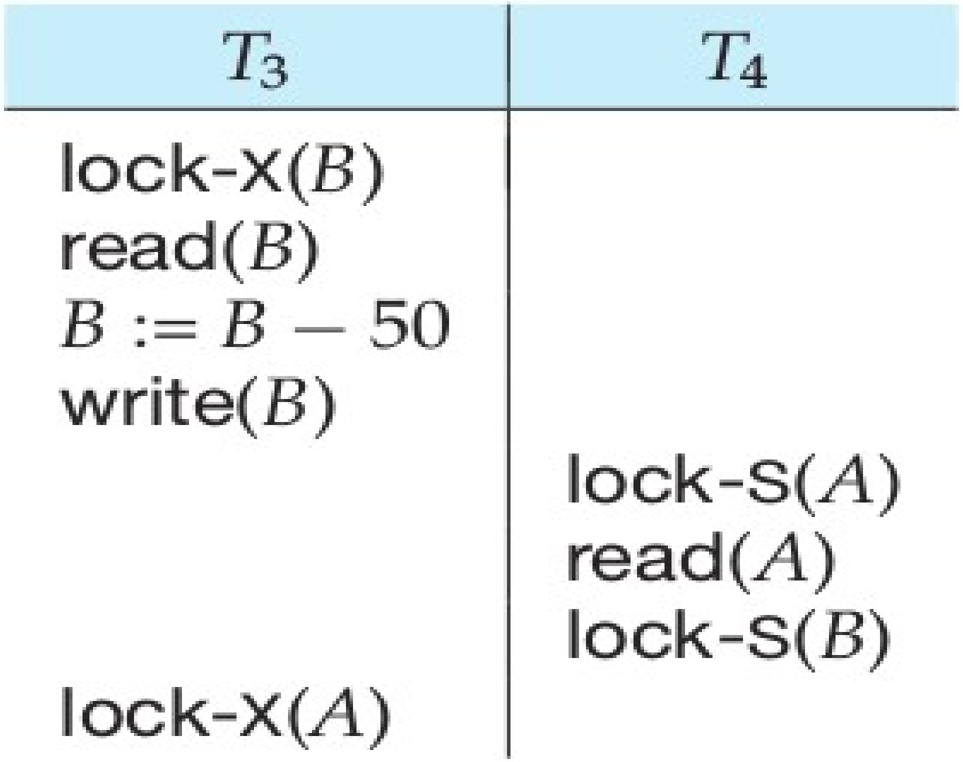


```
$T_3$: lock-X($B$);
       read($B$);
       $B := B - 50$;
       write($B$);
       lock-X($A$);
       read($A$);
       $A := A + 50$;
       write($A$);
       unlock($B$);
       unlock($A$).
```



```
$T_4$: lock-S($A$);
       read($A$);
       lock-S($B$);
       read($B$);
       display($A + B$);
       unlock($A$);
       unlock($B$).
```

Granting of Locks



T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Granting of Locks

- Request of lock is granted only if, no other transaction is holding lock in an incompatible mode, on requested data item.
- Starvation Condition can occur

- When a transaction Ti request a lock on data item Q in a particular mode M, the lock is granted provided that
  - **There is no other transaction holding a lock on Q in a mode incompatible with M.**
  - **There is no other transaction that is waiting for a lock on Q and that made its lock request before Ti.**

Two-Phase Locking Protocol

- One protocol that ensures serializability is the two-phase locking protocol.
- This protocol requires that each transaction issue lock and unlock requests in two phases:

Growing phase: A transaction may obtain locks, but may not release any lock.

Shrinking phase: A transaction may release locks, but may not obtain any new locks.

Two-Phase Locking Protocol

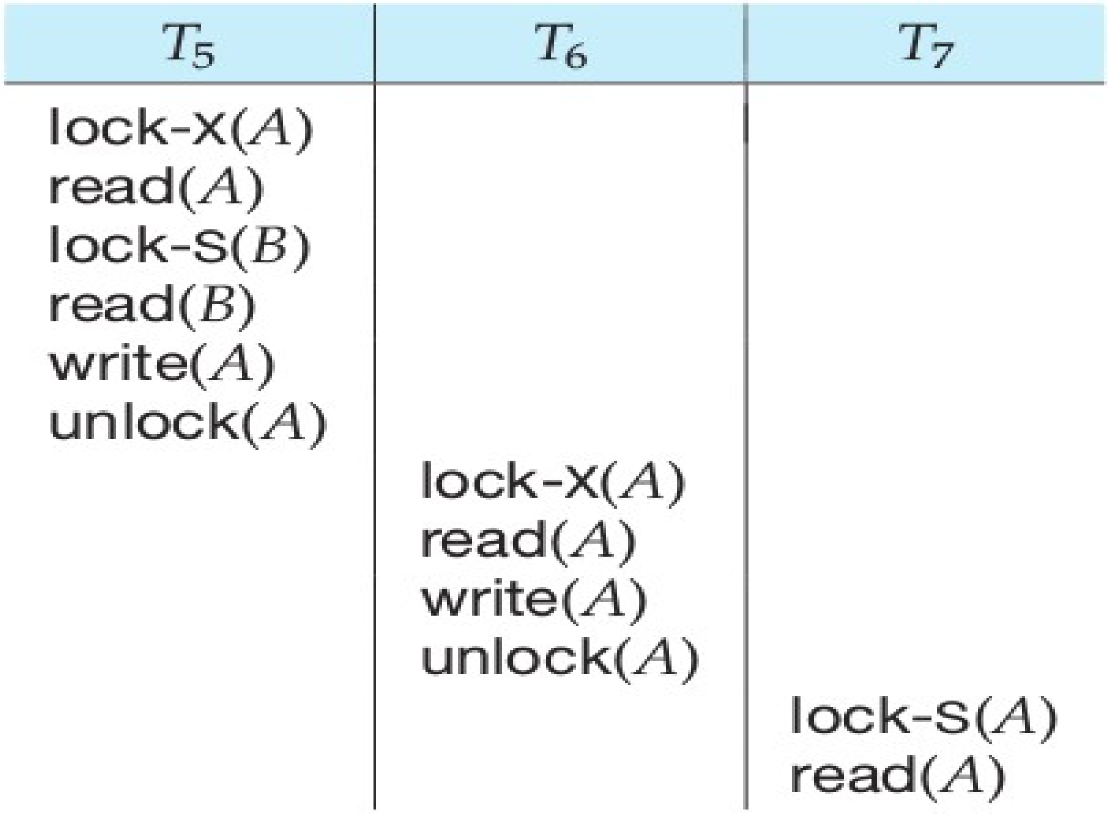
- Initially, a transaction is in the growing phase.
- The transaction acquires locks as needed.
- Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests.
- The point in the schedule where the transaction has obtained its final lock (the end of its growing phase) is called the lock point.

Lock Conversions

- Two-phase locking with lock conversions:
  - Upgrading:
    - can convert a lock-S to a lock-X (upgrade)
  - Downgrading:
    - can convert a lock-X to a lock-S (downgrade)

- Upgrading can take place in only growing phase, whereas downgrading can take place in only the shrinking phase.
- This protocol assures serializability. But still relies on the programmer to insert the various locking instructions

Limitations of Two phase locking protocol



T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

Modified Two-Phase Locking Protocol

- **Strict two-phase locking** - Here a transaction must hold all its exclusive locks till it commits/aborts.

- **Rigorous two-phase locking** is even stricter: here all locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

Timestamp Based Protocols

- With each transaction Ti, system associate a unique fixed timestamp, denoted by TS(Ti)

- This timestamp is assigned before Ti starts execution

- If a transaction Ti has been assigned a timestamp TS(Ti), and a new transaction Tj enters the system,

  TS(Ti)< TS(Tj)

- Methods for implementing –

➢ system clock

➢ logical counter

Timestamp Based Protocols

Each data item Q, associate with two timestamp values

- W-timestamp(Q) – denotes the largest timestamp of any transaction that executed **write(Q)** successfully.
- R-timestamp(Q) – denotes the largest timestamp of any transaction that executed **read(Q)** successfully.
- These timestamp are updated whenever a new read(Q) or write(Q) instruction is executed.

Timestamp Ordering Protocol

- Suppose that transaction Ti issues read(Q)

- If TS(Ti) < W-timestamp(Q), then Ti needs to read a value of Q that was already overwritten. Hence, the read operation is rejected, and Ti is rolled back.

- If TS(Ti) ≥ W-timestamp(Q),then the read operation is executed, and R-Timestamp(Q) is set to the maximum of R-Timestamp(Q) and TS(Ti).

Timestamp Ordering Protocol

- Suppose that transaction Ti issues write(Q)

- If TS(Ti) < R-timestamp(Q), then Ti needs to update a value of Q that was already read by other transaction. Hence, the write operation is rejected, and Ti is rolled back.

- If TS(Ti) < W-timestamp(Q),then Ti is attempting to write an obsolete value of Q. Hence the system rejects this write operation and rolls Ti back.

- Otherwise, the system executes write(Q) operation and sets W-Timestamp(Q) to TS(Ti).

Deadlock Handling

- System is deadlocked if there is a set of waiting transactions such that every transaction in the set is waiting for another transaction in the set.

- Strategies to handle deadlock

- Deadlock Prevention
- Deadlock Detection
- Deadlock Recovery

Deadlock Prevention

Deadlock prevention protocols ensure that the system will never enter into a deadlock state.

Some prevention strategies :

- Require that each transaction locks all its data items before it begins execution (predeclared).
- Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order .

Deadlock Prevention

- **wait-die scheme** — non-preemptive
  - older transaction may wait for younger one to release data item. Younger transactions never wait for older ones; they are rolled back instead.

- **wound-wait scheme** — preemptive
  - older transaction wounds (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.

Deadlock Prevention

- **Timeout-Based Schemes :**

- a transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
- thus deadlocks are not possible
- simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.

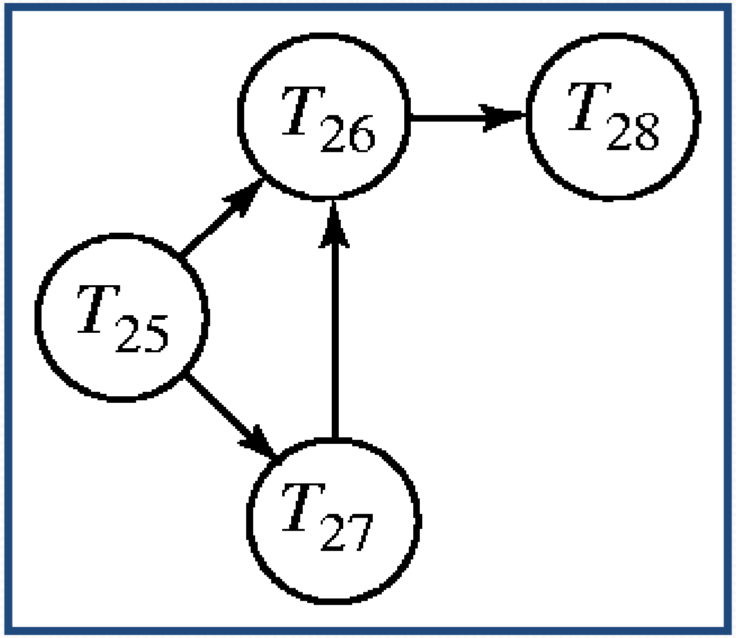
Deadlock Detection

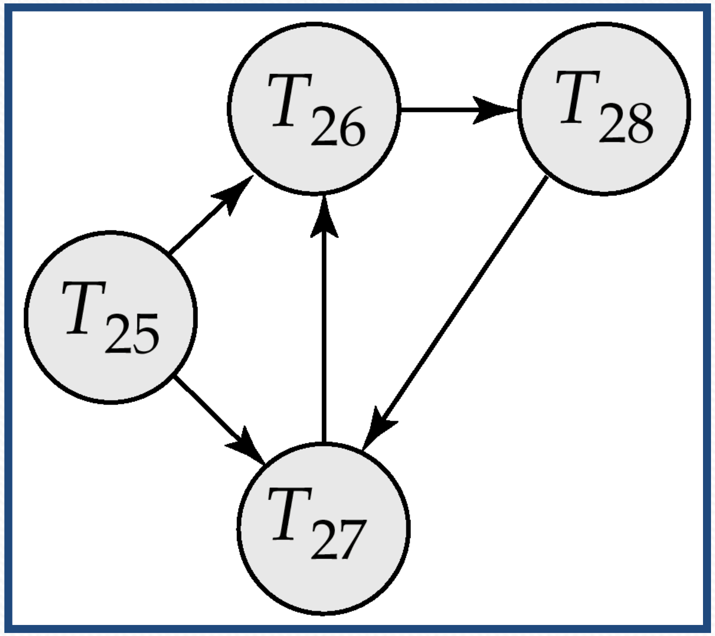
Deadlocks can be described as a wait-for graph, which consists of a pair G = (V, E),

- V is a set of vertices (all the transactions in the system)
- E is a set of edges; each element is an ordered pair Ti →Tj.

- If Ti → Tj is in E, then there is a directed edge from Ti to Tj, implying that Ti is waiting for Tj to release a data item.

Deadlock Detection





- The system is in a deadlock state if and only if the wait-for graph has a cycle.
- Must invoke a deadlock-detection algorithm periodically to look for cycles.

Deadlock Recovery

- When deadlock is detected :

- Select a victim
- Rollback –
  - a. Total Rollback
  - b. Partial Rollback
- Starvation

Recovery System

- Recovery system can restore the database to the consistent state that existed before the failure.

- Recovery system must provide high availability; that is, it must minimize the time required for recovery.

- Failure Classification
  - Transaction Failure – Logical Error , System Error
  - System Crash
  - Disk Failure

Recovery Methods

- To recover from transaction failure following recovery schemes are used -

- Log based recovery

- Shadow paging

Log-Based Recovery

- Log is the most widely used structure for recording database modifications.
- The log is a sequence of log records, recording all the update activities in the database.

- Types of Log -
- ➢ Start Log Record - < Ti, start >
- ➢ Update Log Record - < Ti, Xj, V1, V2 >
- ➢ Commit Log Record - < Ti, commit >
- ➢ Abort Log Record - < Ti, abort >

Log-Based Recovery

- Whenever a transaction performs a write, a log record for that write is created.

- Once a log exists, we can output the modification to the database if that is desirable.

- Database modification types -
  - Deferred Database Modification
  - Immediate Database Modification

- This scheme have the ability to undo a modification if failure occurs

Deferred Database Modification

- The deferred modification technique ensures transaction atomicity by recording all database modifications in the log, but deferring the execution of all write operations of a transaction until transaction partially commits.
- When a transaction partially commits, the information in the log associated with transaction is used in executing deferred writes.
- If failure/abort of transaction occurs then log is simply ignored by system.
- If failure occur after execution then redo(Ti) is performed
- **redo(Ti)** – It sets the value of all data items updated by transaction Ti to the new values.

Deferred Database Modification

Example -	Log	Database
T0: Read(A)	<T0,start>	
A = A – 50		
Write(A)	<T0,A,1000,950>	
Read(B)		
B = B+50		
Write(B)	<T0,B,2000,2050>	
		A = 950
		B = 2050
	<T0,commit>	
T1: Read(C)	<T1, start>	
C = C-100		
Write(C)	<T1,C,700,600>	
		C = 600
	<T1,commit>	

Deferred Database Modification

Failure occur – Write(B)

Example -	Log
T0: Read(A)	<T0,start>
A = A – 50	
Write(A)	<T0,A,1000,950>
Read(B)	
B = B+50	
Write(B)	<T0,B,2000,2050>

In above case as no commit log record appears in the log system will ignore the log.

Deferred Database Modification

Failure occur – Write(C)

Example -	Log
T0: Read(A)	<T0,start>
A = A – 50	
Write(A)	<T0,A,1000,950>
Read(B)	
B = B+50	
Write(B)	<T0,B,2000,2050>
	<T0,commit>
T1: Read(C)	<T1, start>
C = C-100	
Write(C)	<T1,C,700,600>

In above case redo(T0) wiil be done as both start and commit of T0 appear in log, and as no commit log record of T1 appears in the log system will ignore the log.

Deferred Database Modification

Failure occur – <T1,commit>

Example -	Log
T0: Read(A)	<T0,start>
A = A – 50	
Write(A)	<T0,A,1000,950>
Read(B)	
B = B+50	
Write(B)	<T0,B,2000,2050>
	<T0,commit>
T1: Read(C)	<T1, start>
C = C-100	
Write(C)	<T1,C,700,600>
	<T1,commit>

In above case redo(T0) and redo(T1) wiil be done as both start and commit of T0 and T1 appears in log.

Immediate Database Modification

- The immediate modification technique allows database modifications to be output to the database while the transaction is still in the active state.

- undo(Ti) – It restores the value of all data items updated by transaction Ti to the old values.

- redo(Ti) – It sets the value of all data items updated by transaction Ti to new values.

Immediate Database Modification

Example -	Log	Database
T0: Read(A)	<T0,start>	
A = A – 50		
Write(A)	<T0,A,1000,950>	A = 950
Read(B)		
B = B+50		
Write(B)	<T0,B,2000,2050>	B = 2050
	<T0,commit>	
T1: Read(C)	<T1, start>	
C = C-100		
Write(C)	<T1,C,700,600>	C = 600
	<T1,commit>	

Immediate Database Modification

Failure occur – Write(B)

Example -	Log
T0: Read(A)	<T0,start>
A = A – 50	
Write(A)	<T0,A,1000,950>
Read(B)	
B = B+50	
Write(B)	<T0,B,2000,2050>

In above case as no commit log record appears in the log system will undo(T0).

Immediate Database Modification

Failure occur – Write(C)

Example -	Log
T0: Read(A)	<T0,start>
A = A – 50	
Write(A)	<T0,A,1000,950>
Read(B)	
B = B+50	
Write(B)	<T0,B,2000,2050>
	<T0,commit>
T1: Read(C)	<T1, start>
C = C-100	
Write(C)	<T1,C,700,600>

In above case redo(T0) wiil be done as both start and commit of T0 appear in log, and as no commit log record of T1 appears in the log system will undo(T1).

Immediate Database Modification

Failure occur – <T1,commit>

Example -	Log
T0: Read(A)	<T0,start>
A = A – 50	
Write(A)	<T0,A,1000,950>
Read(B)	
B = B+50	
Write(B)	<T0,B,2000,2050>
	<T0,commit>
T1: Read(C)	<T1, start>
C = C-100	
Write(C)	<T1,C,700,600>
	<T1,commit>

In above case redo(T0) and redo(T1) wiil be done as both start and commit of T0 and T1 appears in log.

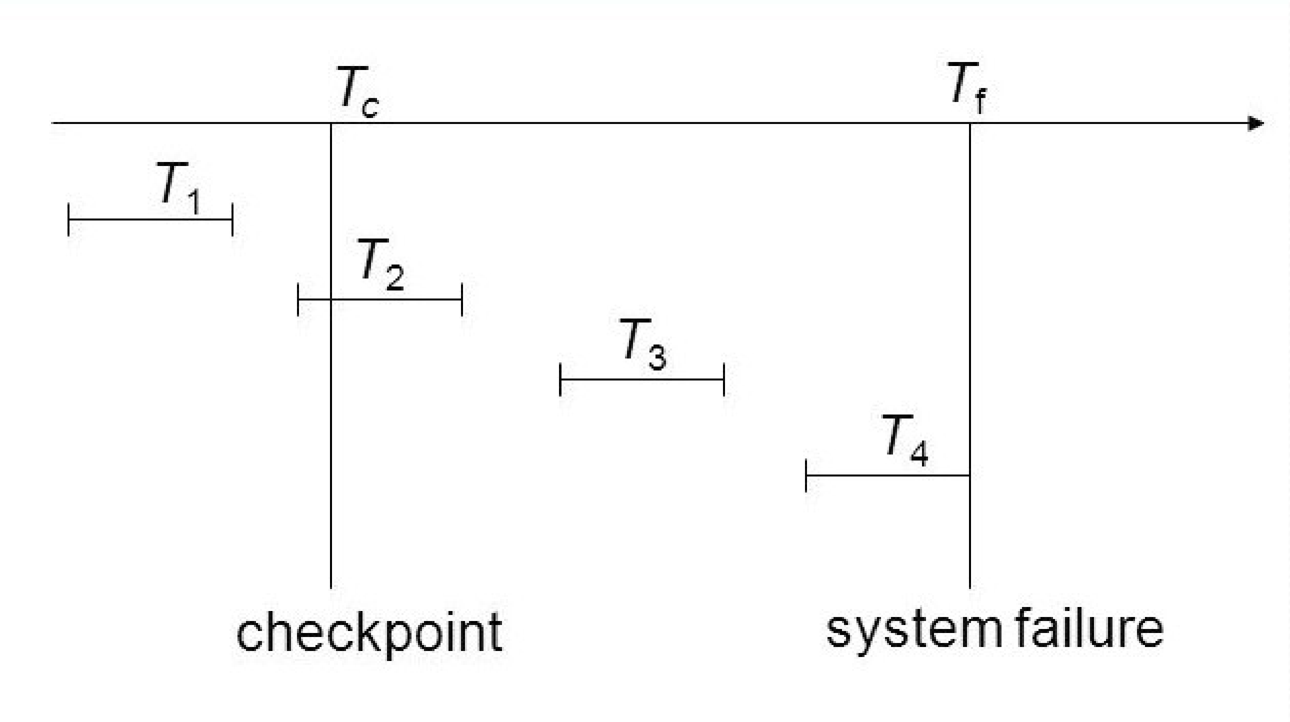
Checkpoints

- Log based recovery required to search entire log to determine which transaction need to be undone or redone.

- Difficulties -
  - The search process is time consuming.
  - Most of the transaction need to be redone, have already written their updates into the database.

- To reduce such types of overhead checkpoints are used.
- System periodically performs checkpoints, log written by system <checkpoint>

Checkpoint Example



- T1 will be ignored (already updated)
- T2 and T3 will be redo
- T4 will be undo

Shadow Paging

- Database is partitioned into fixed size blocks- pages
- Location of all pages is stored – page table

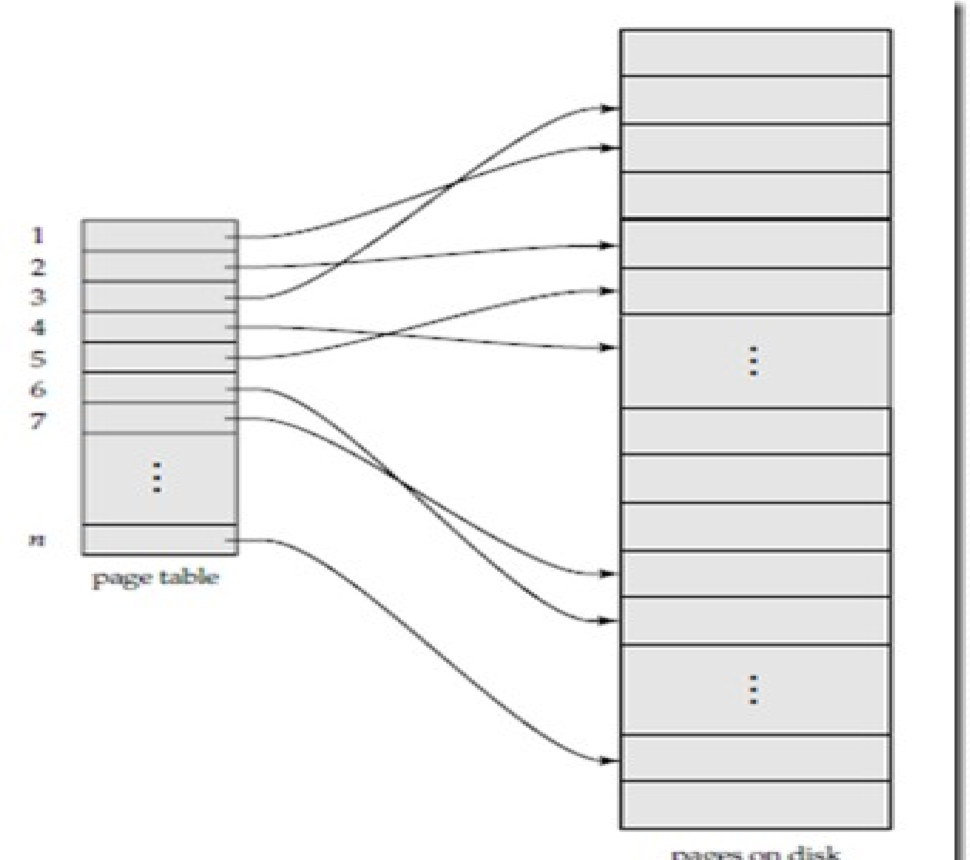


Figure 17.8 Sample page table.

Shadow Paging

- Shadow paging technique maintains two tables
- Current page table and Shadow page table

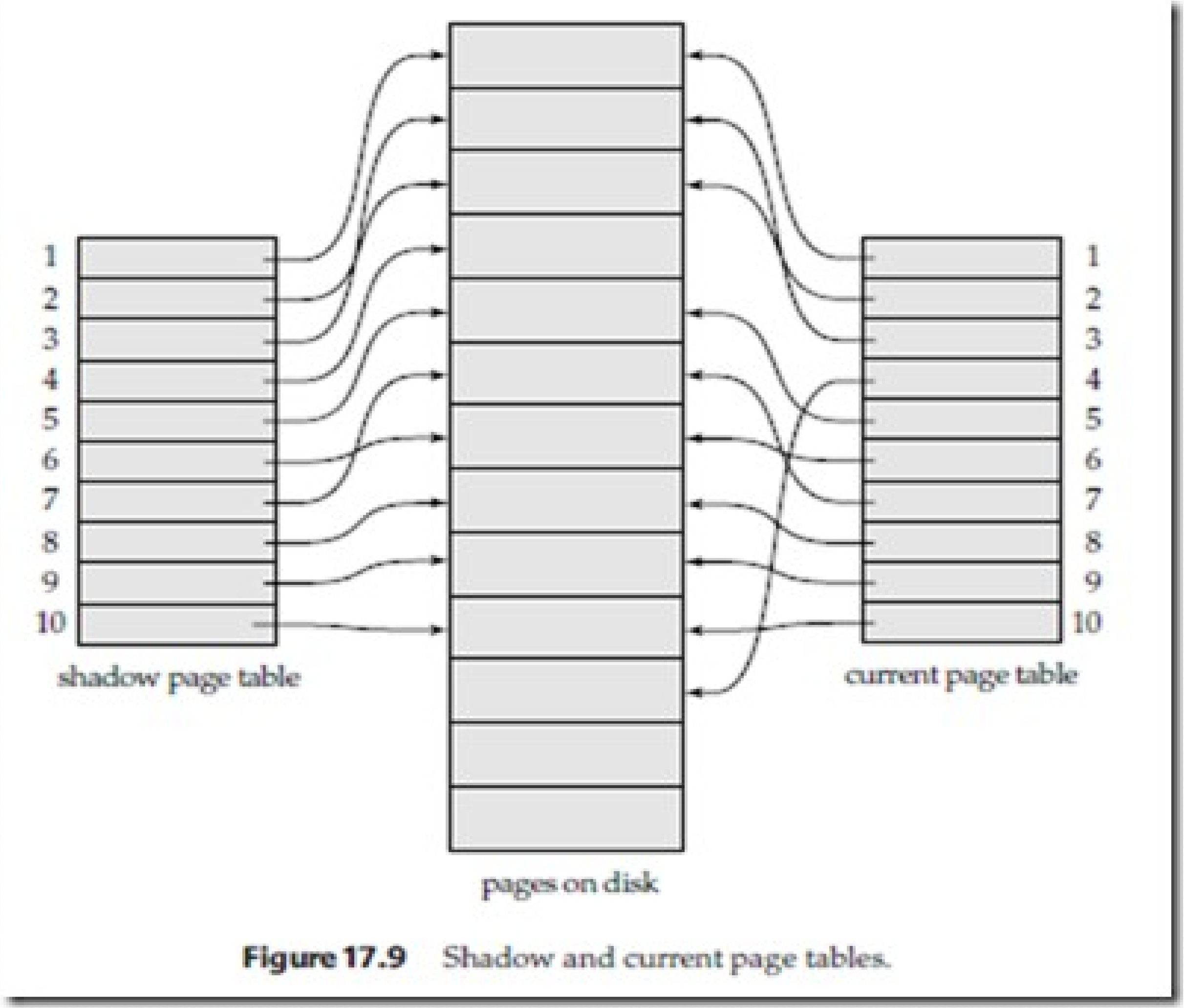
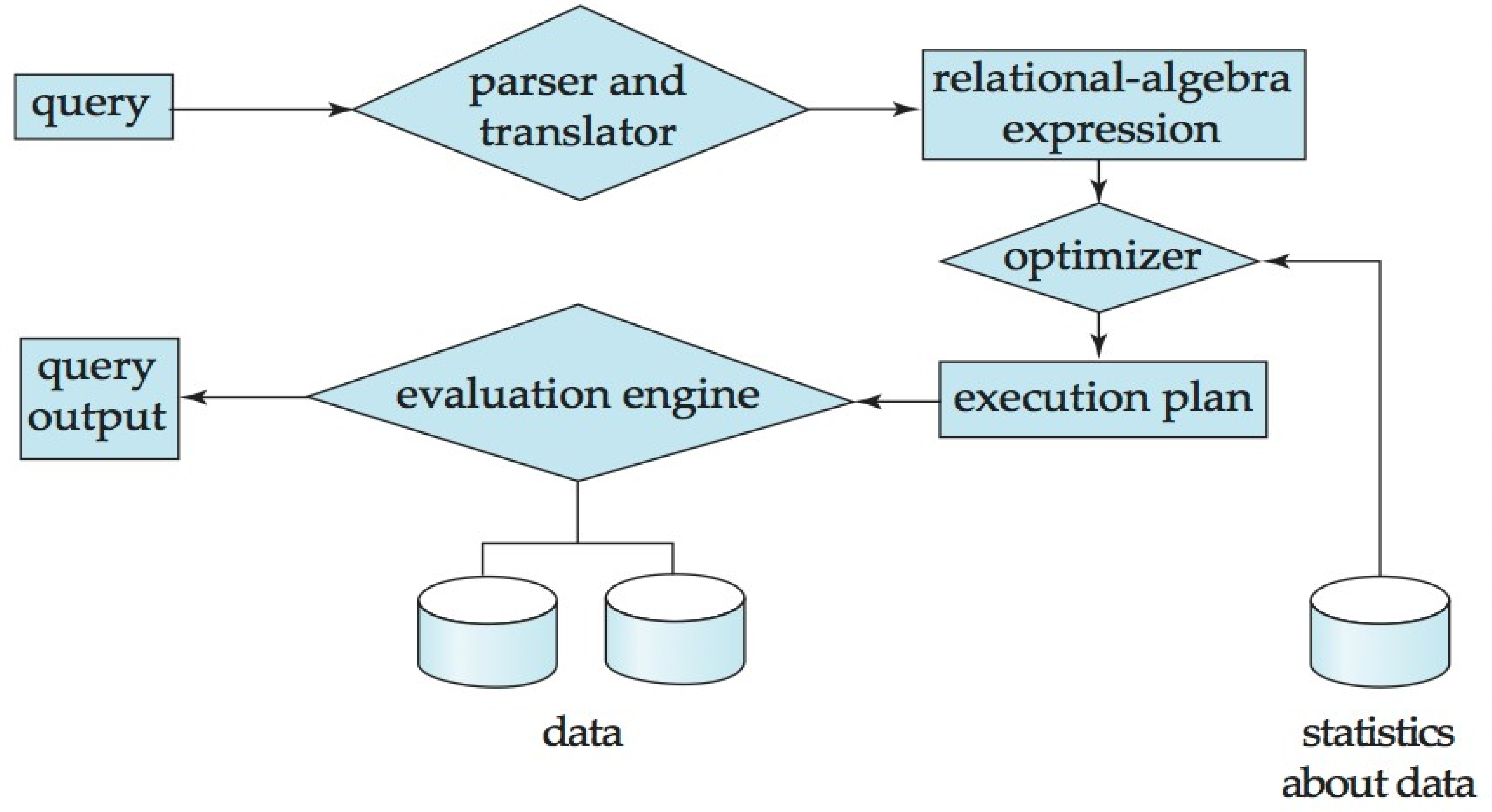


Figure 17.9 Shadow and current page tables.

Query Processing,Optimization and Performance tuning



Thank You