UNIT – 6

Transaction Management

Outline....

- Transaction concepts
- Properties of transactions
- Serializability of transactions
- Testing for serializability
- System recovery
- Two- Phase Commit protocol
- Recovery and Atomicity
- Log-based recovery
- Concurrent executions of transactions and related problems
- Locking mechanism
- Solution to concurrency related problems
- Deadlock
- Two-phase locking protocol
- Isolation
- Intent locking

Transaction Concept

- **Collection of operations that from a single logical unit of work are called transaction.**
- A database system must ensure proper execution of transaction despite failure. It must manage concurrent execution of transactions to avoid inconsistency.

Properties of Transaction

- **Atomicity**: Either all operation of the transaction are reflected properly in the database, or non are.
- **Consistency**: Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation**: Even through multiple transactions execute concurrently, each transaction is unaware of there transaction executing concurrently in the system.
- **Durability**: After a transaction completes successfully, the changes it has made to the database persist, even if there are system failure.
- These properties are all so called **ACID** properties.

Transactions access data using two operations

i. read (x), which transfer the data item x from the database to local buffer belonging to the transaction that executed the read operation.

ii. write (x), which transfer the data x from the local buffer to the transaction that executed the write back to database.

- Example: Let Ti be a transaction that transfer $50 from account A to account B.
- This transaction can be defined as:

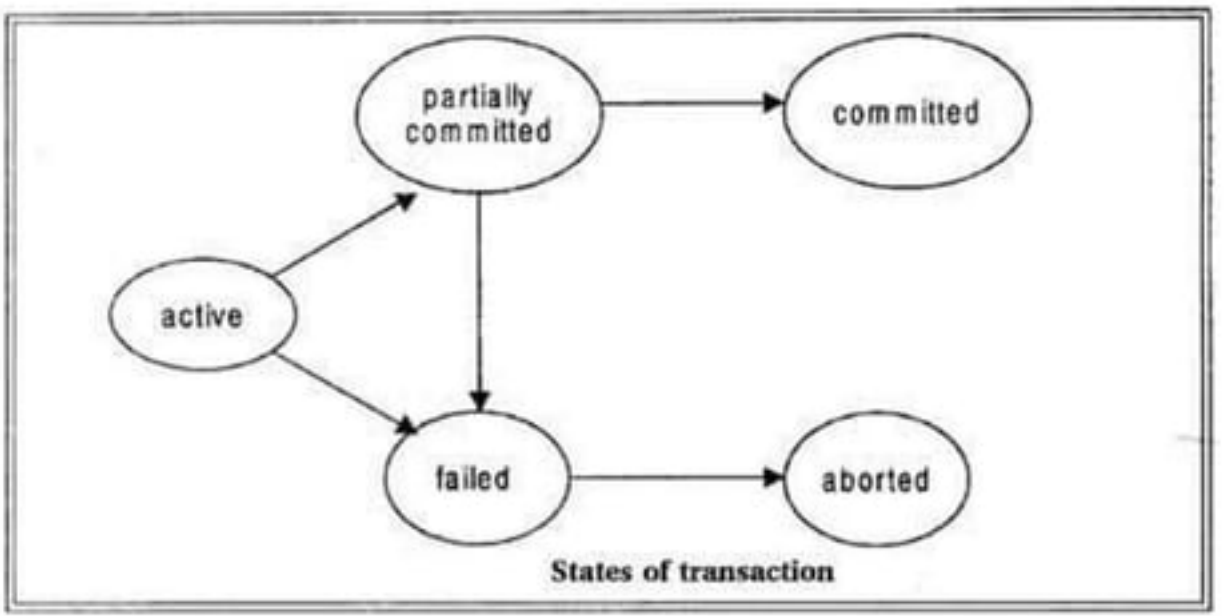
➢ Ti : read(A)

A: = A – 50
write (A)
read (B)
B: = B + 50
write (B)

Transaction State

- In the absence of failure, all transaction complete execution successfully. However , it is also possible that transaction does not complete its execution successfully. Such a transaction is termed as **aborted**.
- A Transaction must be in one of the following states:
- **Active**, the initial sate; 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 database has been restored to its state prior to the start of the transaction.
- **Committed**, after successful completion.

Transaction State



States of transaction

Transaction State

- A transaction starts in the active state. When it finishes its final statement, it enters the partial committed state.
- When the last statement of the actual output is written out in disk from main memory, the transaction enter the committed sates.
- A transaction enter the failure state after the system determine that the transaction can no longer process with its normal execution. Such a transaction must be roll back.
- Then, it enters the aborted state. At this point the system has two option:

1. It can restart the transaction, if the transaction was aborted as a result of some hardware or software error that was created through the internal logic of the transaction.
2. It can be kill the program, if the transaction was aborted because of internal and logical error that can be corrected only by rewriting the application program.

Concurrent Execution of Transaction and Related Problems

- Transaction processing system usually allow multiple transaction to run concurrently. Concurrently execution of multiple transaction cause several complications with consistency of data.

1. **Improved throughput and resource utilization**

"Throughput is number of transaction executed in a given amount of time."

- The transaction consists of many steps. The CPU and the I/O system can operate in parallel. The parallelism of the CPU and the I/O system can therefore be exploited to run multiple transactions in parallel. If one transaction is reading or writing data on disk, another can be running in the CPU. Thus the processor and disk spend less time idle.

2. **Reduce waiting time**

- If transaction run serially, a short transaction may have to wait for a preceding long transaction to compete, which can lead to unpredictable delays in running a transaction.
- Concurrent execution reduces the unpredictable delays in running transaction.

Example

- Let T1 and T2 are two transactions. Transaction T1, transfer $50 from account A to account B. it is defined as:
- T1: read(A)

  A: = A – 50;

  write(A);

  read(B);

  B: = B + 50;

  write(B);
- Transaction T2, transfer 10% of the balance from account A to account B. it is defined as:
- T2: read(A)

  temp: = A * 0.1;

  A: = A – temp;

  write(A);

  read(B);

  B: = B + temp;

  write(B);

Example

- Suppose the concurrent values of account A and B are $1000 and $2000, respectively.
- Suppose the two transactions are executed in order T1 and T2.

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)

Example

- The final value of account A and B, after the execution of schedule 1 are $855 and $2145 respectively.
- If the two transaction are executed in the order T2 followed by T1, then

T1	T2
	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

- After the execution of schedule 2 the sum A + B is preserved, and the final values of account A and B are $850 and $2150 respectively.
- The execution sequence which represent the chronological order in which instructions are executed in the systems, are called **schedules**.
- Given two transactions can be executed concurrently.

T1	T2
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

- Not all concurrent executions result in a correct state. Consider a schedule show in fig after the execution of this schedule, we arrive at a state where the final values of accounts A and B are $950 and $ 2100, respectively. The final state is in consistent state.

T1	T2
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)

- We can ensure consistency of the database under concurrent execution by making sure that any schedule that executed has the same effect as that of serial schedules.

Serializability of Transaction

- **Need of Serializability**

➢Concurrent execution have following problems.

1. **Lost updates**: The update of one transaction is overwritten by another transaction.

Example: Suppose T1 credit $100 to account A and T2 debit $50 from account A. The initial values of A = 500. If credit are applied correctly, then final correct value of the account should be 550. If we run T1 and T2 concurrently as follows:

T1 (Credit)	T2 (Debit)
read (A) {A = 500}	read (A) {A = 500}
A: A + 100 {A = 600}	A: A - 50 {A = 450}
write (A) {A = 600}	write (A) {A = 450}

- Final value of A= 450. The credit of T1 is missing (lost Update) from the account.

Serializability of Transaction

2. **Dirty read**: Reading of a non-existent value of A by T2. If T1 updates A which is then ready by T2, then if T1 aborts T2 will have read a value of A which never existed.

T1 (Credit)	T2 (Debit)
read (A) {A = 500}	
A: A + 100 {A = 600}	
write (A) {A = 600}	
	read (A) {A = 500}
T1 failed to complete	A: A + 100 {A = 600}
	write (A) {A = 600}

- T1 modified A = 600. T2 read A = 600. But T1 failed and its effect is removes from the database, so A is restored to its old value, i.e. A = 500.
- A=600 is nonexistent value but read (reading dirty data) by T2.

Serializability of Transaction

3. **Unrepeatable read**: If T2 reads A, which is then altered by T1 and T1 commits. When T2 rereads A it will find different values of A in its second read.

T1 (Credit)	T2 (Debit)
read (A) {A = 500}	read (A) {A = 500}
A: A + 100 {A = 600}	A: A - 50 {A = 450}
write (A) {A = 600}	write (A) {A = 600}

- In this execution T1 reads A=500, T2 read A=500. T1 modifies A to 600. when T2 rereads A it gets A=600. This should not be the case. T2 in the same execution should get only one value of A (500 or 600 and not both).
- In serial execution these problems would not arise since serial execution does not share data items.
- This means we can use the results of serial execution as a measure of correctness and concurrent execution for improving resource utilization.
- We need **serialization of concurrent** transaction.

Serializability of Transaction

- **Serialization of concurrent transactions:** Process of managing the execution of a set of transactions in such a way that their concurrent execution produces the same end result as if they were run serially.
- **Definition of Serialization**
- Given an interleaved execution of a set of n transactions; the following conditions hold for each transaction in the set.

1. All transactions are correct in the sense that if any one of the transactions is executed by itself on a consistent database, the resulting database will be consistent.
2. Any serial execution of the transaction is also correct and preserve the consistency of the database; the results obtained are correct.

- Types of Serializability

1. **Conflict serializability**
2. **View serializability**

Conflict Serializability

- Let us consider a schedule S in which there are two consecutive instructions Ii and Ij of transactions Ti and Tj respectively ($i \neq j$).
- If Ii and Ij refer to different data items, then we can swap Ii and Ij without affecting the results of any instruction in the schedule.
- However, if Ii and Ij refer to the same data item Q, then the order of the two steps may matter.
- There are four case to consider:

1. Ii = read(Q), Ij = read(Q). The order of Ii and Ij does not matter, since the same value of Q is read by Ti and Tj regardless of the order.
2. Ii = read(Q), Ij = write(Q). If Ii comes before Ij, then Ti does not read the value of Q that is written by Tj in instruction Ij. If Ij comes before Ii, then Ti read the value of Q that is written by Tj. Thus the order of Ii and Ij matters.

Conflict Serializability

3. Ii =write (Q), Ij = read (Q). The order of Ii and Ij matters, reason is same as previous case.
4. Ii = write(Q), Ij = write(Q). Since both instruction are write operations, the order of these instructions does not affect either Ti or Tj. However, the value obtained by the next read(Q) instruction of S is effected, since the result of only the letter of the two write instructions is preserved in the database.

- Thus, only in the case where both Ii and Ij are read instructions, the order of execution does not matter.
- We say that Ii and Ij **conflict** if they are operations by different transactions on the same data item and at least one of these instructions is a **write** operation.

Schedule 3 – Showing only the read and write operations

T1	T2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

- The write(A) of T1 conflict with read(A) of T2. However, write (A) of T2 does not conflict with read(B) of T1, hence we can swap these instructions to generate a new Schedules 5. Regardless of initial system state, Schedule 3 and 5 generates same result.

Schedule 5 – Schedule 3 after swapping a pair of instruction

T1	T2
read (A)	
write (A)	
	read (A)
read (B)	
	write (A)
write (B)	
	read (B)
	write (B)

Schedule 6 – A serial schedule that is equitant to schedule 3

- We can continue to swapping nonconflict instructions:
- Swap the read(B) instruction of T1 with read (A) instruction of T2.
- Swap the write(B) instruction of T1 with write (A) instruction of T2.
- Swap the write(B) instruction of T1 with read (A) instruction of T2.

T1	T2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

View Serializability

- Consider two schedules S and S', where the same set of transactions participates in both schedules. The schedules S and S' are said to be view equivalent if three conditions are met:

1. For each item of Q, if transaction Ti, reads the initial value of Q in schedule S, then transaction Ti must, in schedules S', also read the initial value of Q.
2. For each data item Q, if transaction Ti executes read(Q) in schedules S, and if that value was produced by a write(Q) operation executed by transaction Tj, then the read(Q) operation of transaction Ti must, in schedule S', also read the value of Q that was produced by the same write(Q) operation of transaction Tj.
3. For each data item Q, the transaction that performs the final write(Q) operation in schedule S must perform the final write(Q) operation in schedule S'.

Schedule 8 – A view serializable schedule

T3	T4	T5
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

- In Schedule 8, transactions T4 and T5 perform write(Q) operations without having performed a read(Q) operation. Write of this sort called blind write.
- View serializable schedule with blind writes is not conflict serializable.

Testing Serializability

- Testing of serializability is done by using directed graph, called **precedency graph**, constructed from schedules.
- This graph consists of a pair G = (V,E), where V is set of vertices and E is a set of edges. The set of vertices consists of all transactions in schedules.
- The set of edges consists of all edges Ti -> Tj for which one of three conditions holds:

1. Ti execute write (Q) before Tj executes read (Q).
2. Ti executes read (Q) before Tj executes write (Q).
3. Ti executes write (Q) before Tj executes write (Q).

Testing Serializability

- The precedence graph for schedule 1 it contains a single edge T1 -> T2, since all the instructions of T1 are executed before the first instruction of T2 is executed.

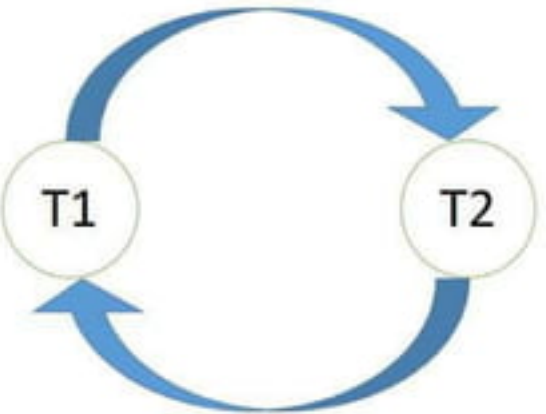
- The precedence graph for schedule 2 it contains a single edge T2 -> T1, since all the instructions of T2 are executed before the first instruction of T2.

Schedule - 9

T1	T2
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)

Testing Serializability

- The precedence graph for schedule 9.



Test for Conflict Serializability

- To test conflict serializability, construct a presentence graph for given schedule. If **graph contain cycle, the schedule is not conflict serializability**. If the graph **contains no cycle, then the schedule is conflict serializable**.
- Schedule 1 and 2 are conflict serializable, as the precedence graph for both schedules does not contain any cycle.
- While the schedule 9 is not conflict serializable, as precedence graph for it contains cycle.

Topological sorting

- If the graph is acyclic, then using topological sorting given below , find serial schedule:

1. Initialize the serial schedule as empty.
2. Find the transaction Ti, such that there are no arcs entering Ti, Tj is the next transaction in the serial schedule.
3. Remove Ti and all edges emitting from Ti. If the remaining set is non-empty, return to step 2, else the serial schedule is complete.

Example: Is the corresponding schedule conflict serializable ?

T9	T10	T11
read (A)		
A: = F1(A)		
write (A)	read (A)	
	A: = f2 (A)	
	write (A)	
	read (A)	
	B: = f3(B)	
	write (B)	

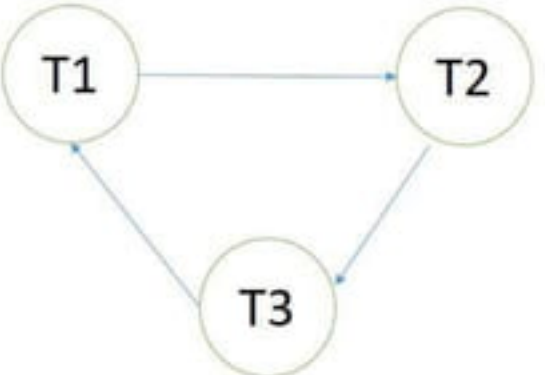
Example: Is the corresponding schedule conflict serializable ?

- As the graph is acyclic, the schedule is conflict serializable.

Example: Is the corresponding schedule conflict serializable ?

T1	T2	T3
Read (A)		
	Read (B)	
A: = f1 (A)		
		Read (C)
	B: = f2(B)	
	Write (B)	
		C:= f3(C)
		Write(C)
Write (A)		
	Read (A)	
	A:= f4 (A)	
Read (C)		
	Write (A)	
C: = f5(C)		
Write (C)		
		B: = f6(B)
		Write (B)

Example: Is the corresponding schedule conflict serializable ?



- As the graph is cyclic, the schedule is non conflict serializable.

Test for View Serializability

- The precedence graph used for testing conflict serializability cannot be used for testing view serializability.
- We need to extend the precedence graph to include labeled edges. This graph is called as **labeled precedence graph**.
- **Construction of labeled precedence graph**
- Lest S be a schedule consisting of transactions {T1,T2,..,Tn}. Let Tb and Tf be two dummy transactions such that Tb issues write (Q) for each Q accessed in S, and Tf issues read(Q) for each Q accessed in S.

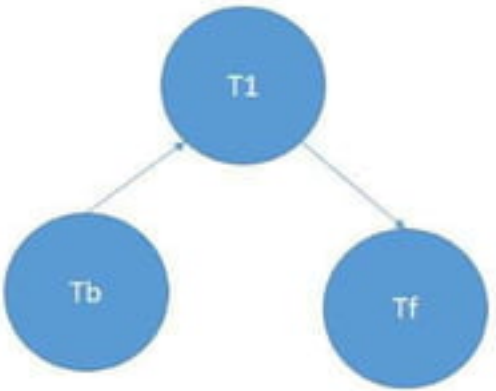
Test for View Serializability

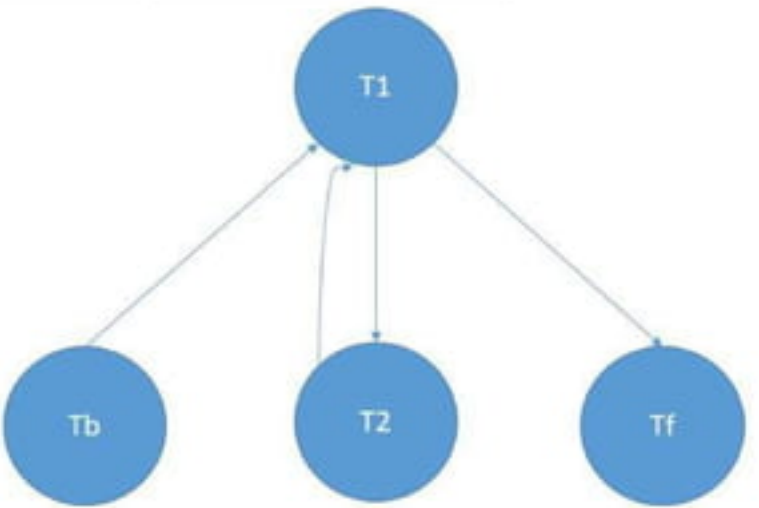
- **Construction of labeled precedence graph**

1. Add an edge Ti → Tj, if transaction Tj reads the value of data item Q write by transaction Ti.
2. Remove all edges incident on useless transactions. A transaction Ti is useless if there exists no path, in the precedence graph from Ti to transaction Tf.
3. For each data item Q such that Tj reads the value Q written by Ti, and Tk executes write (Q) and Tk ≠ Tb, do following :
    1. If Ti = Tb and Tj ≠ Tf, then insert the edge Tj → Tk in the labeled precedence graph.
    2. If Ti ≠ Tb and Ti = Tf, then insert the edge Tk→Ti in the labeled precedence graph.
    3. If Ti ≠ Tb and Tj ≠ Tf, then insert the pair of edges Tk→Ti and Tj→Tk where p is a unique integer larger larger than 0 that has not been used earlier for labeling edges.

Example: Prepare the labeled precedence for following schedule?

T1	T2
read (Q)	
	write (Q)
write (Q)	





Recoverability

- If a transaction Ti fails, we need to undo the effect of this transaction to ensure the atomicity property of transaction.
- In a system that allows concurrent execution, it is necessary to ensure that any transaction Tj that is dependent on Ti should also be aborted.
- To achieve this surety, we need to place restriction on the type of schedules permitted in the system.
- Types of schedules that are acceptable from the view of recovery from transaction failure are:

1. **Recoverable schedules**
2. **Cascadeless schedules**

Recoverable schedules

- A recoverable schedule is one where, for each pair of transaction Ti and Tj such that Tj reads a data item previously written by Ti, the commit operation of Ti appears before the commit operation of Tj.

T8	T9
read (A)	
write (A)	
	read (A)
read (B)	

- Consider schedule in which T9 is a transaction that perform only one instruction: read (A). Suppose that the system allows T9 to commit immediately after executing the read(A) instruction.
- Thus, T9 commits before T8 does.

Recoverable schedules

- Suppose that T8 fails to before it commits. Since T9 has read the value of data item.
- A written by T8, we must abort T9 to ensure transaction atomicity.
- However, T9 has already committed and cannot be aborted. Thus, it is impossible to recover correctly from the failure of T8.
- Thus, this schedule is no recoverable schedule, which should not be allowed.

Cascadeless schedules

- Even if a schedule is recoverable, to recover correctly from the failure of a transaction Ti, we may have to roll back several transaction. Such situation occur if transactions have read data written by Ti.

T1	T2	T3
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)

Cascadeless schedules

- Consider schedule Transaction T1 writes a value of A that is read by transaction T2. Transaction T2 writes a value of A that is read by T3. Suppose that, at this point, T1 fails, T1 must be roll back.
- Since T2 dependent on T1, T2 must be rolled back . Similarly as T3 is dependent on T2, T3 should also be rolled back.
- This phenomenon, in which a single transaction failure leads to a series of transaction roll back, is called cascading rollback.
- Cascading rollback is undesirable, since it leads to the undoing of a significant amount of work. Therefore, schedules should not contain cascading rollbacks. Such schedules are called cascade less schedules.
- A **cascade less schedule** is one where, for each pair of transaction 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.

Concurrency Control

- The system must control the interaction among the concurrent transaction. This control is achieved through one of the concurrency control scheme.
- The concurrency control schemes are based on the serializability property.
- Different types of protocol/schemes used to control concurrent execution of transactions are:

1. Lock Based Protocol
2. Timestamp Based Protocol

Lock Based Protocol

- To ensure serializability, it is required that data items should be accessed in mutual exclusive manner; if one transaction is accessing a data item, no other transaction can modify that data item.
- To implement this requirement locks are used.
- A transaction is allowed to access a data item only if it is currently holding a lock on that item.
- **Locks**

1. **Shared mode lock:**
   - If a transaction Ti has obtained a shared mode lock on item Q, then Ti can read, but cannot write Q. It is denoted by S.
2. **Exclusive:**
   - If a transaction Ti has obtained an exclusive mode lock on item Q, then Ti can read and also write Q. It is denoted by X.

Lock Based Protocol

- A transaction can unlock a data item Q by unlock(Q) instruction.

	S	X
S	True	False
X	False	False

- Example:
  - Transaction display the total amount of money in accounts A and B.

```
Lock-S(A);
Read(A);
Unlock(A);
Lock-S(B);
Read(B);
Unlock(B);
Display (A+B);
```

Locking Protocols

- Each transaction in the system should follow a set of rules, called a locking protocol, indicating when a transaction may lock and unlock each of data items.
- **Granting of locks:**
- When a transaction requests a lock on a data item in a particular mode and no other transaction has a lock on the same data item in a conflict mode, the lock can be granted.
- Starvation of transaction can be avoided by granting lock in the following manner.

1. There is no other transaction holding a lock on Q in a mode that conflict with M.
2. 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

- This protocol requires that each transaction issue lock and unlock requests in two phase.

1. **Growing phase**
   - In this phase, a transaction may obtain locks, but may not released any lock.
2. **Shrinking phase**
   - In this phase, a transaction may release locks, but may not obtain any new locks.

- Initially, a transaction is in the growing phase. The transaction acquires lock as needed. Once the transaction release a lock, it enters in the shrinking phase, and it cannot issue more lock requests.

Two Phase Locking Protocol

- Example:

```
Lock-X(B);
Read(B);
B:= B – 50;
Write (B);
Lock-X(A);
Read(A);
A:= A + 50;
Write(A);
Unlock(B);
Unlock(A);
```

Two Phase Locking Protocol

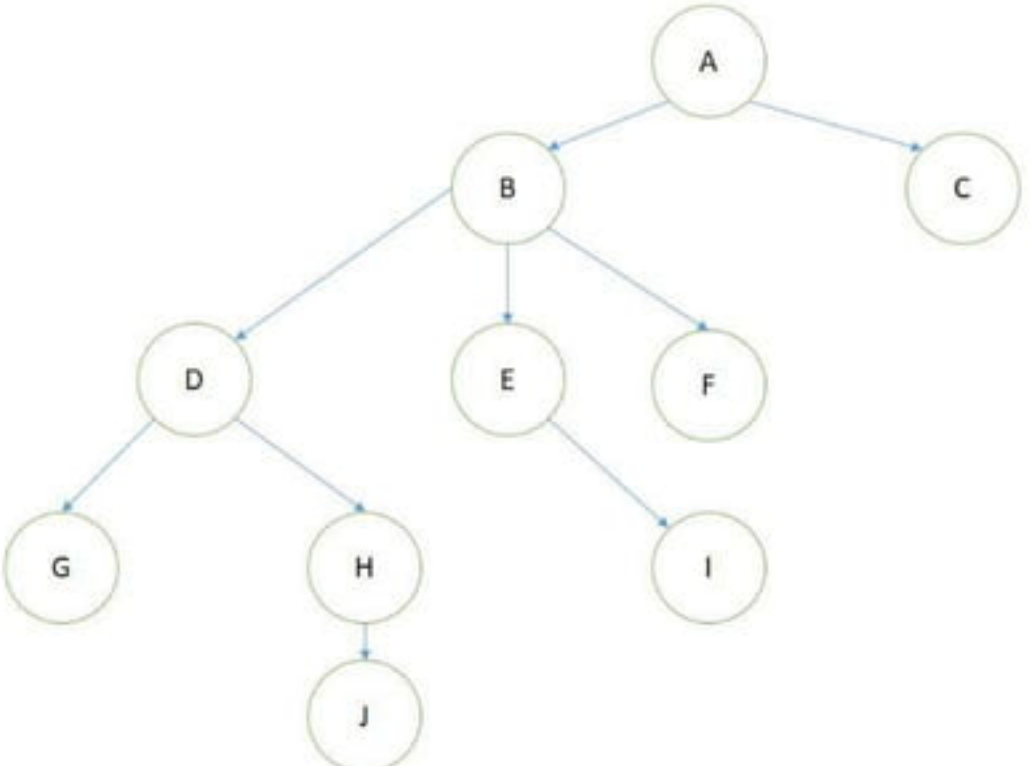
- **Advantage**
- The two-phase locking protocol ensures conflict serializability. Consider any transaction.
- The point in the schedule where the transaction has obtained its final lock is called the lock point of the transaction.
- Now, transaction can be ordered according to their lock points. This ordering is a serializability ordering for the transaction.
- **Disadvantages**
- Two phase locking does not ensure freedom from deadlock.
- Casacading rollbacks may occur under two-phase locking.
  - Strict two-phase locking protocol
  - The rigorous two-phase locking protocol

Graph Based Protocol

- One of the example of graph based protocol is **tree protocol.**
- In the tree protocol, the only allowed lock instruction is lock-X.
- Each instruction must observe the following rules:

1. The first lock by Ti may be on any data item.
2. Subsequently, a data item Q can be locked by Ti only if the parent or Q is currently locked by Ti.
3. Data item may be unlocked at any time.
4. A data item that has been locked and unlocked by Ti cannot subsequently be relocked by Ti.

Graph Based Protocol



Timestamp Based Protocol

- Time stamp based protocol ensures serializability. It selects an ordering among transactions in advance using time stamps.
- Each transaction in the system, a unique fixed timestamp is associated it is denoted by TS(Ti).
- If a transaction Ti has been assigned timestamp TS(i) and a new transaction Tj enters the system, then TS(Ti) < TS(Tj).
- Two method are used for implementing timestamp:

1. Use the value of the system clock as timestamp.
2. Use a logical counter.

Timestamp Based Protocol

- To implement this scheme, two timestamps are associated with each data item Q

i. **W-timestamp(Q)** denotes the largest timestamp of any transaction that execute write (Q) successfully.

ii. **R-timestamp(Q)** denotes the largest timestamp of any transaction that read(Q) successfully.

- These timestamp are updated whenever a new read(Q) or write(Q) instruction is executed.

Timestamp Ordering Protocol

- The timestamp ordering protocol ensures that any conflicting read and write operations are executed in timestamp order.

1. Suppose that transaction Ti issues read(Q).

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

B. 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(i).

2. Suppose that transaction Ti issues write (Q).

A. If TS(Ti) < R-timestamp(Q), then the value of Q that Ti is producing was needed previously , and the system assumed that the value would never be produced. Hence, the system rejects write operation and rolls Ti back.

B. 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 back Ti.

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

- If a transaction Ti is rolled by the concurrency control scheme, the system assigned it a new timestamp and restarts it.

Thoma's Write Rule

- Thoma's write rule is modified version of timestamp ordering protocol.
- Consider schedule given in following fig.

T16	T17
Read(Q)	
	Write (Q)
Write (Q)	

- Here, T16 start before T17, therefore TS(T16) < TS(T17). The read(Q) operation of T16 succeeds, similarly the write(Q) operation of T17. when T16 attempts is write(Q) operation, it is rejected by the system and T16 is rollback as TS(T166) < W-timestamp(Q). Since W-timestamp(Q) = TS (T17).

Thoma's Write Rule is

1. Suppose that transaction Ti issues write (Q).

A. If TS(Ti) < R-timestamp(Q), then the value of Q that Ti is producing was needed previously , and the system assumed that the value would never be produced. Hence, the system rejects write operation and rolls Ti back.

B. If TS(Ti) < W-timestamp(Q), then Ti is attempting to write an obsolete value of Q. Hence, this write operation can be ingnored.

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

Deadlock

- **Definition of Deadlock**
  - " A system is in a deadlock state if there exist a set of transactions such that every transaction in the set is waiting for another transaction in the set. "
- There are two principal method for dealing with the deadlock problem

1. **Deadlock prevention** : This approach ensure that system will never enter in deadlock state.
2. **Deadlock detection and recovery:** This approach tries to recover from deadlock if system enters in deadlock state.

Deadlock prevention

- There are two approaches for deadlock prevention:

1. One approach ensures that no cyclic waits can occur by ordering the request for locks, or requiring all locks to be acquired together. This approach required that each transaction locks all data items before it begins execution.

- **Disadvantages of this approach are :**

a) It is hard to predict before the transaction begins, what data items need to be locked.

b) Data-item utilization may be very low, since many of the data items may be locked but unused for a long time.

2. The second approach for deadlock prevention is to use preemption and transaction rollback. In preemption when a transaction T2 requests a lock that transaction T1 holds, the lock granted to T1 may be preempted by rolling back T1, and granting of lock to T2. The system uses timestamp to decide whether a transaction should wait or roll back.

Deadlock prevention

- Two different deadlock prevention schemes using timestamp are.

1. **Wait die**

- The wait-die scheme is no preemption technique. In this, when transaction Ti requests a data item held by Tj, Ti is allowed to wait only if it has a timestamp smaller than Tj. Otherwise , Ti is rolled back (dies).

2. **Wound wait**

- The wound-wait is preemptive technique. In this, when transaction Ti requests data item held by Tj, Tj is allowed to wait, only if it has timestamp greater than Tj (Ti is younger than Tj). Otherwise Tj rolled back.

Timeout Based Schemes

- This approach for deadlock handling is based on lock timeouts.
- In this approach, a transaction that has request a lock waits for at most a specified amount of time. If the lock has not been granted within that time, the transaction is said to be time out, and it rolls back itself and restarts.

Deadlock Detection and Recovery

- This approach use an algorithm that examines the state of the system periodically to determine whether a deadlock has occurred. If one has, then the system attempts to recover from the deadlock.
- Deadlock can be described in terms of directed graphs called **wait-for graph.** This graph consists of a pair G = < V, E >.
- Example:
- Transaction T25 is waiting for Transaction T26 and T27.
- Transaction T27 is waiting for Transaction T26.
- Transaction T26 is waiting for Transaction T28.

Recovery from Deadlock

- Three actions need to be taken:

1. **Selection of transactions:**

- Given a set of deadlock transactions, we must determine which transactions should be rolled back to break the deadlock. The method used for this is: rollback those transaction that will incur **minimum cost**.

i. How long the transaction has compute and how much longer the transaction will compute before it completes its task.

ii. How many data items the transaction has used.

iii. How many more data items the transaction need to complete its task.

iv. How many transaction will be involved in the rollback.

Recovery from Deadlock

2. **Rollback**

- Once we have decided to roll back particular transaction, we must determine how far transaction should be roll back. The solution are:

i. **Total rollback:** Abort the transaction and then restart it.

ii. **Partial rollback:** It is more effective to roll back the transaction only far as necessary to break the deadlock.

Recovery from Deadlock

3. **Starvation**

- It is possible that, same transaction will be rollback number of times to break the deadlock. As a result, this transaction never completes its designated task. Thus there is starvation. To avoid this, we must ensure that transactions can be picked as a victim only a small number of times.

System Recovery

- There are various types of failure that may occur in a system:

1. **Transaction failure :**
   a) **Logical error :** Logical error occurs because of some internal condition, such as bad input, data not found, overflow or resource limit exceeded.
   b) **System error :** Example of system error id deadlock.
2. **System crash :**

- There is a hardware malfunction, or a bug in the database software or in the operating system, that cause the loss of the content of volatile storage and brings transaction processing to a halt.

3. **Disk failure :**

- A disk block lose its content as a result of either a head crash or failure during a data transfer operation.

Recovery Schemes

- To recover from transaction failure following recovery schemes are used

1. **Log-based recovery**
2. **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 updates in the database.

a) **Update log record :** It describe is single database write. It has following fields

- **Transaction identifier** is the unique identifier of the transaction that perform the write operation.
- **Data item identifier** is the unique identifier of the data item written. Typically, it is the location on the disk of the data item.
- **Old value** is the value of the data item prior to the write.
- **New value** is the value of the data item that it will have after the write.

Log-based recovery

- Various types of log records are represents as:
- **< Ti start >** : Transaction Ti has started.
- **< Ti, Xj, V1, V2>** : Transaction Ti has perform a write on data item Xj. Xj had value V1 before the write, and will have value V2 after the write.
- **< Ti commit >** : Transaction Ti has committed.
- **< Ti abort >** : Transaction Ti has aborted.

Log-based recovery

- Two technique that use log to ensure transaction atomicity despite failures are:

1. **Deferred Database Modification**
2. **Immediate Database Modification**

Log-based recovery - Deferred Database Modification

- The deferred modification technique ensure transaction atomicity by according all database modification in the log, but deferring the execution of all write operations on a transaction until transaction partially commits.
- **Example:**

1. Consider two transactions T0 and T1. Transaction T0 transfer $50 from account A to B.
2. Let transaction T1 withdraws $100 from account C

Log-based recovery - Deferred Database Modification

- T0: read (A);
  A: = A – 50;
  write (A);
  read (B);
  B: = B + 50;
  write(B);
- T1: read(C);
  C: = C – 100;
  write (C);
- Suppose that these transactions are executed serially, in the order T0 followed by T1, and the values of account A, B and C before the execution are $1000, $2000 and $700 respectively.

Log-based recovery - Deferred Database Modification

< T0 start >

< T0 , A , 950 >

< T0 , B , 2050 >

< T0 commit >

< T1 start >

< T1 , C , 600 >

< T1 commit >

Log-based recovery - Deferred Database Modification

Log	Database
< T0 start >	
< T0 , A , 950 >	
< T0 , B , 2050 >	
< T0 commit >	
	A= 950
	B= 2050
< T1 start >	
< T1 , C , 600 >	
< T1 commit >	
	C= 600

Log-based recovery - Deferred Database Modification

- Using the log, the system can handle any failure that results in the loss of information on volatile storage. The recovery scheme are uses the following recovery procedure :
- **redo (Ti) :** It sets the value of all data items updates by transaction Ti to the new values.
- The set of data items updates by Ti and their respective new values of can be found in the log.

Log-based recovery - Deferred Database Modification

(a)	(b)	(c)
< T0 start >	< T0 start >	< T0 start >
< T0 , A , 950 >	< T0 , A , 950 >	< T0 , A , 950 >
< T0 , B , 2050 >	< T0 , B , 2050 >	< T0 , B , 2050 >
	< T0 commit >	< T0 commit >
	< T1 start >	< T1 start >
	< T1 , C , 600 >	< T1 , C , 600 >
		< T1 commit >

Log-based recovery - 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.
- **Example:**

1. Consider two transactions T0 and T1. Transaction T0 transfer $50 from account A to B.
2. Let transaction T1 withdraws $100 from account C

- Suppose that these transactions are executed serially, in the order T0 followed by T1, and the values of account A, B and C before the execution are $1000, $2000 and $700 respectively.

Log-based recovery - Immediate Database Modification

< T0 start >

< T0 , A , 1000, 950 >

< T0 , B , 2000, 2050 >

< T0 commit >

< T1 start >

< T1 , C , 700, 600 >

< T1 commit >

Log-based recovery - Immediate Database Modification

Log	Database
< T0 start >	
< T0 , A , 1000, 950 >	
< T0 , B , 2000, 2050 >	
	A= 950
	B= 2050
< T0 commit >	
< T1 start >	
< T1 , C , 700, 600 >	
	C= 600
< T1 commit >	

Log-based recovery - Immediate Database Modification

- Using the log, the system can handle any failure that results in the loss of information on volatile storage. The recovery scheme are uses the following recovery procedure :
- **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 the new values.
- The set of data items updates by Ti and their respective new values of can be found in the log.

Log-based recovery - Immediate Database Modification

(a)	(b)	(c)
< T0 start >	< T0 start >	< T0 start >
< T0 , A , 1000, 950 >	< T0 , A , 1000, 950 >	< T0 , A , 1000, 950 >
< T0 , B , 2000, 2050 >	< T0 , B , 2000, 2050 >	< T0 , B , 2000, 2050 >
	< T0 commit >	< T0 commit >
	< T1 start >	< T1 start >
	< T1 , C , 700, 600 >	< T1 , C , 700, 600 >
		< T1 commit >

Log-based recovery - Checkpoints

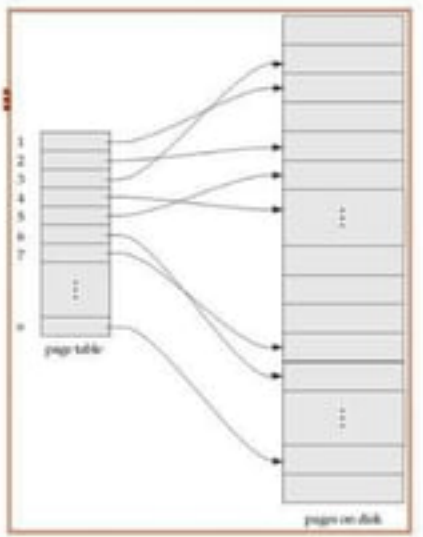
- When a system failure occurs, the log is consulted to determine which transactions need to be undone and which to be redone. For doing this, it is necessary to search the entire log. There are two difficulties with this approach :

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

- To reduce these types of overheads, checkpoints are used.
- For all transactions Tk in T that have no < Tk commit > recode in the log, executed undo (Tk).
- For all transaction Tk in T such that the record < Tk commit > appear in the log executed redo (Tk).
- **Undo operation is not applied when deferred modification technique is used.**

Shadow Paging recovery

- **The database is partitioned into some number of fixed length blocks, which are referred to as pages.**
- The pages are sorted in any random order on disk. Therefore there should be some way to find out Ith pages of the database for any given I.
- For this purpose, page table is used.



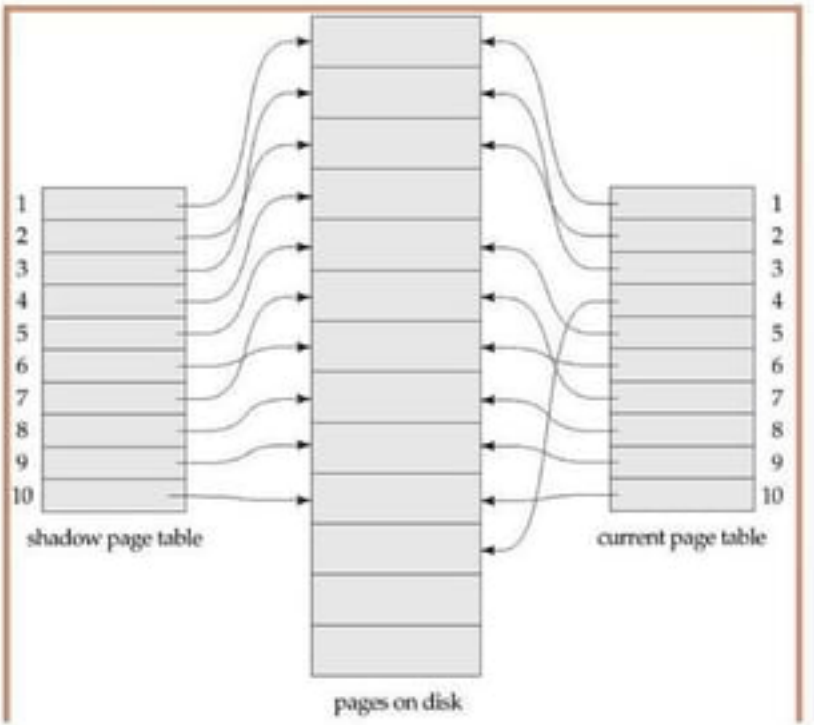
Shadow Paging recovery

- A sample page table is show in fig. The key idea behind the shadow paging technique is to maintain two page table during the life of a transaction :

1. Current page table
2. Shadow page table

- When the transaction starts, both page tables are identical. The shadow page table is never change over the duration of the transaction.
- The current page table may be change when a transaction performs a write operation. All input and output operations use the current page table to locate database page on disk.

Shadow Paging recovery



Shadow Paging recovery

- **Advantages**

1. Shadow paging requires fewer disk accesses than do the log based recovery.
2. The overhead of log record output is eliminated.
3. Recovery from crashes is significantly faster as no undo and redo operation is required.

- **Disadvantages**

1. **Commit overhead**
2. **Data fragmentation**
3. **Garbage collection**

Isolation

- In database system, isolation is a property that defines how/when the changes made by one operation become visible to other concurrent operations.
- **Isolation Levels**
- The isolation levels are important in that, they can assure the developer of the validity of the data read and updated during a transaction.
- There are four DBMS transaction isolated levels:

1. **Serializable Read**
2. **Repeatable Read**
3. **Read committed**
4. **Read uncommitted**

Isolation

1. **Serializable Read**

- This isolation level specifies that all transaction occur in a completely isolated fashion; i.e. as if all transactions in the system has executed serially, one after the other.
- The serializable transaction isolation level assured that:

i. SELECT queries issued during transaction cannot read data that has been modified but not yet committed by other transaction.

ii. Other transaction cannot update/alter the data that has been read by the current transaction until the current transaction completes.

iii. Other transaction cannot insert new rows with key values that would fall in the range of keys read by any statements in the current transaction until the current transaction complete.

Isolation

2. Repeatable read

- All data records read by a SELECT statement cannot be change; however, if the SELECT statement contains any ranged WHERE clause.
- The Repeatable read transaction isolation level assured that:

i. SELECT queries issued during transaction cannot read data that has been modified but not yet committed by other transaction.

ii. Other transaction cannot update/alter the data that has been read by the current transaction until the current transaction completes.

Isolation

3. Read committed

- In this isolation level, read locks are acquired on selected data but they are released immediately whereas write locks are released at the end of the transaction.
- The Repeatable read transaction isolation level assured that:

i. SELECT queries cannot read data that has been modified but not yet committed by other transaction.

ii. The data read by one transaction can be changed by other transactions between individual statements within the current transaction.

Isolation

4. **Read uncommitted**

- In this isolation level, dirty read are allowed. Once transaction may see uncommitted changes made by some other transaction.
- The Read uncommitted transaction isolation level:

i. SELECT queries cannot read data that has been modified but not yet committed by other transaction.

Intent Locking

- In addition to shared, exclusive and update locks, the DBMS also provides another lock know as intent lock.
- Intent locks are placed on higher level database objects when a user of process takes locks on the data pages or rows.
- An intent lock stays in place for the life of the lower-level locks.
- Intent locks are used primarily to ensure that one process cannot taken locks on a table, or pages in the table, that would conflict with the locking of another process.