Chapter One

Transaction Management

And

Concurrent Control

Transaction:-

- Is a sequence of operations that form a single unit of work
- Is a logical unit of work on a database
  - an entire program
  - a portion of a program
  - a single command
- The entire series of steps necessary to accomplish a logical unit of work
- A collection of operations that form a single logical unit of work.
- For instance, transfer of money from one account to another is a transaction consisting of two updates, one to each account.
- Successful transactions change the database from one CONSISTENT STATE to another (One where all data integrity constraints are satisfied)
- A transaction is often initiated by an application program
  - ✓ **begin** a transaction

    - START TRANSACTION
  - ✓ **end** a transaction

    - COMMIT (if successful) or ROLLBACK (if errors)

- 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.
- Either the whole transaction must succeed or the effect of all operations has to be undone(rollback)

4 PROPERTIES OF A TRANSACTION

1. **Atomic** – All or Nothing

   All parts of the transaction must be completed and committed or it must be aborted and rolled back

2. **Consistent**

   Each user is responsible to ensure that their transaction (if executed by itself) would leave the database in a consistent state

3. **Isolation**

   - It indicates that action performed by a transaction will be hidden from outside the transaction until the transaction terminates.
   - the database system must take special actions to ensure that transactions operate properly without interference from concurrently executing database statements.

4. **Durability**

   - If a transaction has been committed, the DBMS must ensure that its effects are permanently recorded in the database (even if the system crashes). We can say simple ACID properties of transaction

TRANSACTION MANAGEMENT WITH SQL

- SQL Statements → Commit / Rollback
- When a transaction sequence is initiated it must continue through all

succeeding SQL statements until:

1. A Commit Statement is Reached
2. A Rollback Statement is Reached
3. The End of the Program is Reached (Commit)
4. The Program is Abnormally Terminated (Rollback)

TRANSACTION LOG

❖ Keeps track of all transactions that update the database

- Record for the beginning of the transaction
- Type of operation (insert / update / delete)
- Names of objects/tables affected by the transaction
- Before and After Values for Updated Fields
- Pointers to Previous and Next Transaction Log Entries for the same transaction
- The Ending of the Transaction (Commit)

❖ Used for recovery in case of a Rollback

TRANSACTION STATE

○ Because failures occurs, transaction are broken up into states to handle various situation.

1.Active: The initial state; the transaction stays in this state until while it is still executing.

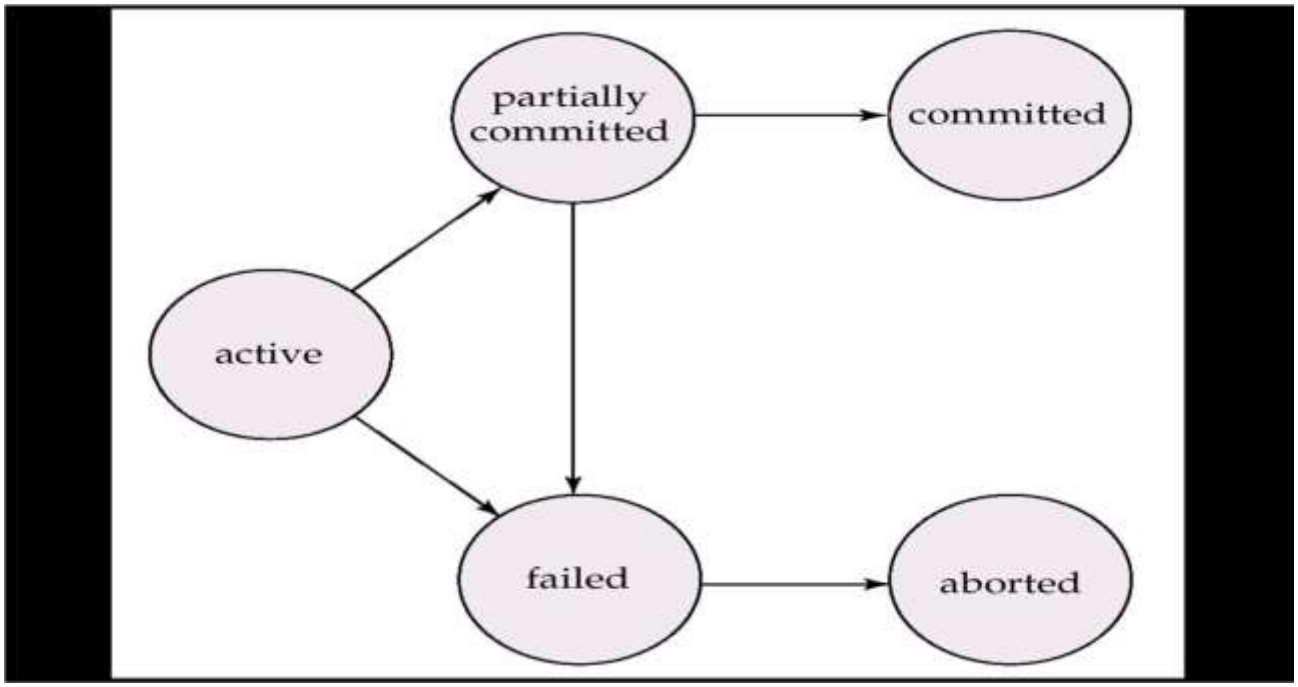
2.Partially Committed: After the final statement has been executed

3.Failed: after the discovery that normal execution can no longer proceed

4.Committed: after successful completion.

5.Aborted: after the transaction has been rolled back the database has been restored to its state prior to the start of the transaction

DIAGRAM OF TRANSACTION STATE



- We say that a transaction has committed only if it has entered the committed state
- Similarly, we say that a transaction has aborted only if it has entered the aborted state.
- A transaction is said to have terminated if it has either committed or aborted.
- A transaction starts in the active state.
- When it finishes its final statement, it enters the partially committed state.
- If the transaction enters the aborted state the system has two options:
  - ✓ It can restart the transaction, but only if the transaction was aborted as a result of some hardware or software error that was not created through the internal logic of the transaction.
  - ✓ It can kill the transaction. It usually does so because of some internal logical error
    - that can be corrected only by rewriting the application program, or

- because the input was bad, or
- Because the desired data were not found in the database.

CONCURRENCY CONTROL

- ✓ Coordination of simultaneous transaction execution in a multiprocessing database system
- ✓ Concurrent access is relatively easy if all transaction are only reading data, as there is no way that they can interfere with one another.
- ✓ However, when two or more transaction is accessing the database simultaneously and at least one is updating data, there may be interference that can result in inconsistencies.
- ✓ Ensure transaction serializability in a multi-user database

Lack of Concurrency Control can create data integrity and consistency problems:

- Lost Updates
- Uncommitted Data

There are two good reasons for allowing concurrency:

1. Improved throughput and resource utilization.

- the number of transactions executed in a given amount of time.
- the processor and disk utilization also increase; in other words, the processor and disk spend less time idle, or not performing any useful work

2. Reduced waiting time.

- There may be a mix of transactions running on a system, some short and some long. If transactions run serially, a short transaction may have to wait for a preceding long transaction to complete, which can lead to unpredictable delays in running a transaction.

- Moreover, it also reduces the average response time:
  - the average time for a transaction to be completed after it has been submitted.

- The motivation for using concurrent execution in a database is essentially the same as the motivation for using multiprogramming in an operating system
- It does so through a variety of mechanisms called concurrency-control schemes

SCHEDULE

- Sequences that indicate 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.
- Consider again the simplified banking system which has several accounts and a set of transactions that access and update those accounts.
- Let T1 and T2 be two transactions that transfer funds from one account to another.

Transaction T1 transfers $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 transfers 10 percent 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).
```

- the sequence of instruction steps is in chronological order from top to bottom

SCHEDULE 1:

A SERIAL SCHEDULE IN WHICH T1 IS FOLLOWED BY T2.

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 in which t2 is followed by t1

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	

- Serial schedule: Schedule that does not interleave the actions of different transactions. Schedule 1 and 2 above are serial

- Recalling a well-known formula from combinatorics, we note that, for a set of n transactions, there exist n factorial (n!) different valid serial schedules.

SCHEDULE 3:

A concurrent schedule equivalent to schedule 1 above.

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

SCHEDULE 4:

- Not all concurrent executions result in a correct state. To illustrate, consider the schedule of Figure follows.

A concurrent schedule may result in an inconsistent state.

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

- 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.
- 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.
- Not all serial schedules are conflict equivalent to each other. For Example, schedules 1 and 2 are not conflict equivalent.

CONCEPT OF SERIALIZABILITY

- Basic Assumption – Each transaction preserves database consistency.
- Thus serial execution of a set of transactions preserves database

consistency.

- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
  1. conflict serializability
  2. view serializability
- We ignore operations other than read and write instructions, and we assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes. Our simplified schedules consist of only read and write instructions.

CONFLICT SERIALIZABILITY

- 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.
- 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

- Example of a schedule that is not conflict serializable:

Schedule 5

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 >.

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

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:

1. For each data item Q, if transaction T_i reads the initial value of Q in schedule S, then transaction T_i must, in schedule S´, also read the initial value of Q.

2. For each data item Q if transaction T_i executes read(Q) in schedule S, and that value was produced by transaction T_j (if any), then transaction T_i must in schedule S´ also read the value of Q that was produced by transaction T_j .

3. For each data item Q, the transaction (if any) that performs the final write(Q) operation in schedule S must perform the final write(Q) operation in schedule S´. As can be seen, view equivalence is also based purely on reads and writes alone.

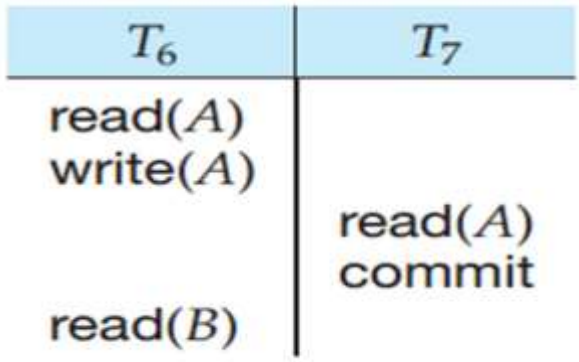
RECOVERABLE SCHEDULE

- Consider the partial schedule 6, in which T7 is a transaction that

performs only one instruction:read (A). We call this a partial schedule because we have not included a commit or abort operation for T6.

Notice that T7 commits immediately after executing the read(A) instruction. Thus, T7 commits while T6 is still in the active state.

Schedule 6 non recoverable schedule



- Now suppose that T6 fails before it commits. T7 has read the value of data item A written by T6. Therefore, we say that T7 is dependent on T6. Because of this, we must abort T7 to ensure atomicity. However, T7 has already committed and cannot be aborted. Thus, we have a situation where it is impossible to recover correctly from the failure of T6.

Schedule 6 is an example of a nonrecoverable schedule.

- A recoverable schedule is one where, 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 commit operation of Tj. For the example of schedule 6 to be recoverable, T7 would have to delay committing until after T6 commits.

CASCADELESS SCHEDULE

Even if a schedule is recoverable, to recover correctly from the failure of a transaction Ti, we may have to roll back several transactions. Such situations occur if transactions have read data written by Ti. As an illustration, consider the partial schedule 7 below.

T_8	T_9	T_{10}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)
abort		

- Phenomenon, in which a single transaction failure leads to a series of transaction rollbacks, is called cascading rollback.
- Cascading rollback is undesirable, since it leads to the undoing of a significant amount of work. It is desirable to restrict the schedules to those where cascading rollbacks cannot occur. Such schedules are called cascadeless schedules.
- Formally, a cascadeless schedule is one where, 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.
- It is easy to verify that every cascadeless schedule is also recoverable

CONCURRENCY CONTROL TECHNIQUES /MECHANISMS/

- There are two main concurrency control techniques that allow transactions to execute safely in parallel subject to certain constraints:

1. **Locking technique**
2. **Timestamp**.

Locking

Lockingtechnique: A lock is a variable associated with each data item that describes the status of the item with respect to possible operations that can be applied to it.

- a lock may deny access to other transactions to prevent incorrect results.

- Types oflock: depending upon types of lock the data manager gives or denies access to other operations on the same data item

BINARY LOCK

BinaryLocks: will have 2 values locked and unlocked represented as 0 and 1.

- A transaction sends a request to a data manager to access to a data item by first locking the data item using LOCK() operation .
- At this moment if any other operation of some other concurrent transaction tries to access same data item then it is forced to wait until the transaction (previous one) that has locked the data item unlock the data item using unlocked command

Few Rules That Must Be Followed

- LOCK() : operation must be issued by transaction before any update operation like read() or write() operation are performed on transaction.
- LOCK() :can't be issued by transaction if already holds LOCK() on data item
- UNLOCK() : must be issued after all read() and write() operations are completed in a transaction.

- UNLOCK(): can't be issued by transaction unless it already hold the lock on the data item.

EXAMPLE FOR BINARY LOCK

- T1:LOCK(A)
- T1:READ(A) A=100
- T1:A=A+200 A=300
- T1:WRITE(A)
- T1:UNLOCK(A)
- T2:LOCK(A)
- T2:READ(A) A=300
- T2:A:=A+300 A=600
- T2:WRITE(A) A=600
- T2:UNLOCK(A)

This example is serializable schedule. Therefore, in case of a binary locking mechanism at most one transaction can hold the lock on a particular data item .Thus no transaction can access the same item concurrently.

LOCK-BASED PROTOCOLS

- A lock is a mechanism to control concurrent access to a data item
  - Data items can be locked in two modes :

1. exclusive (X) mode :

- If a transaction has an exclusive lock on a data item, it can both read and update the item.
- as long as a transaction holds the exclusive lock on the item, no other transactions can read or update that data item

X-lock is requested using lock-X instruction.

- It is also called a write lock

2. shared (S) mode :

- If a transaction has a shared lock on a data item, it can read the item but not update it.
- S-lock is requested using lock-S instruction.
- Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

Lock-compatibility matrix

	S	X
S	true	false
X	false	false

A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions

- Any number of transactions can hold shared locks on an item, but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.

EXAMPLE OF A TRANSACTION PERFORMING LOCKING:

- T1
- LOCKX(A)
- READ(A) A=1000
- A:=A-200 A=800
- WRITE(A) A=800
- UNLOCK(A)
- LOCKX(B)
- READ(B) B=900
- B:=B+200 B=1100
- WRITE(B) B=1100
- UNLOCK(B)

- T2
- LOCKX(SUM)
- SUM:=0
- LOCKS(A)
- READ(A) A=800
- SUM:=SUM+A SUM=800
- UNLOCK(A)
- LOCKS(B)
- READ(B) B=1100
- SUM:=SUM+B SUM=1900
- WRITE(SUM) SUM=1900
- UNLOCK(B)
- UNLOCK(SUM)
- IF EXECUTED SERIALLY THE OUTPUT WILL BE 1900

But If Running Concurrently Then Sum Will Be 2100 /Loss Update

- T2:LOCKX(SUM)
- T2:SUM:=0
- T2:LOCKS(A) A=1000
- T2:READ(A)
- T2:SUM:=SUM+A SUM=1000
- T2:UNLOCK(A)
- T1:LOCKX(A)
- T1:READ(A) A=1000
- T1:A:=A-200 A=800
- T1:WRITE(A) A=800
- T1:UNLOCK(A)
- T1:LOCKX(B)
- T1:READ(B) B=900
- T1:B:=B+200 B=1100
- T1:WRITE(B) B=1100
- T1:UNLOCK(B)
- T2:LOCKS(B)
- T2:READ(B) B=1100
- T2:SUM:=SUM +B SUM=1100
  T2:WRITE(SUM) SUM=1100
- T2:UNLOCK(B)
- T2:UNLOCK(SUM)
- AFTER TRANSACTION SUM+B IS 2100 .This is inconsistent. This locking technique didn't solve this problem because value of a/c A was added to sum before modification was

CONSIDER ANOTHER EXAMPLE

T1

- LOCKX(B)
- READ(B) B=200
- B:=B-50 B=150
- WRITE(B) B=150
- UNLOCK(B)
- LOCKX(A)
- READ(A) A=100
- A:=A+50 A=150
- WRITE(A) A=150
- UNLOCK(A)

T2:

- LOCKS(A)
- READ(A) A=150
- UNLOCK(A)
- LOCKS(B)
- READ(B) B=150
- UNLOCK(B)
- DISPLAY(A+B) A+B=300
- THIS IS CLEAR THAT IF THEY RUN SEQUENTIALLY THE OUT PUT WILL BE 300

When run concurrently

T1

- LOCKX(B)
- READ(B) B=200
- B:=B-50 B=150
- WRITE(B) B=150
- UNLOCK(B)

T2

- LOCKS(A)
- READ(A) A=100
- UNLOCK(A)
- LOCKS(B)
- READ(B) B=150
- UNLOCK(B)
- DISPLAY(A+B)

A+B=250

T1

- LOCKX(A)
- READ(A) A=100
- A:=A+50 A=150
- WRITE(A) A=150
- UNLOCK(A)
- THIS IS AN INCONSISTENT STATE OUTPUT WILL BE 250

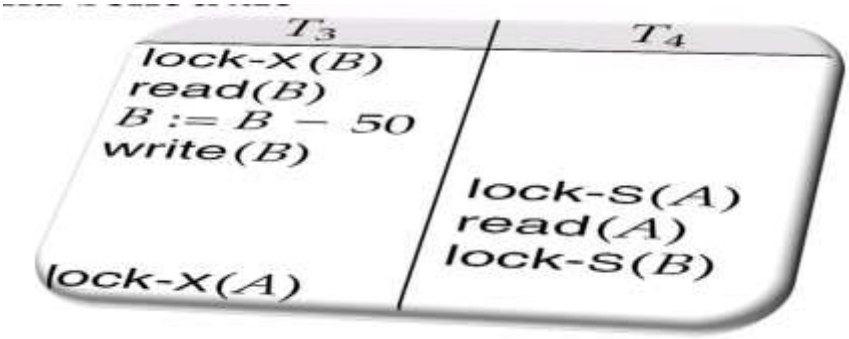
NOW IF WE DELAY UNLOCKING TO THE END OF TRANSACTION THEN

- T1
- LOCKX(B)
- READ(B) B=200
- B:=B-50 B=150
- WRITE(B) B=150
- LOCKX(A)
- READ(A) A=100
- A:=A+50 A=150
- WRITE(A) A=150
- UNLOCK(B)
- UNLOCK(A)

- T2
- LOCKS(A)
- READ(A) A=150
- LOCKS(B)
- READ(B) B=150
- UNLOCK(A)
- UNLOCK(B)
- DISPLAY(A+B)
- HERE ALSO A+B WILL BE 300

PITFALLS OF LOCK-BASED PROTOCOLS

- Consider the partial schedule



- Neither T_3 nor T_4 can make progress — executing lock-S(B) causes T_4 to wait for T_3 to release its lock on B, while executing lock-X(A) causes T_3 to wait for T_4 to release its lock on A.
- Such a situation is called a deadlock. To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.

DEAD LOCK:

- In other words a dead lock occurs when each transaction in a set of 2 or more transaction is waiting for some data item which is locked by other transaction in a set i.e two transaction are waiting for a condition that will never occur.
- The main reason for deadlock is exclusive locks acquired on data item by a transaction

Starvation

- Starvation is also possible if concurrency control manager is badly designed. For example:
- A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
- The same transaction is repeatedly rolled back due to deadlocks.
- For an example T2 has a shared mode lock on data item A and T1 request an exclusive lock on data item A,T1 can't be given that lock until T2 releases shared lock on data item A .But if T3 request a shared lock on A it will be given .

At this moment if T2 releases its lock on A, still T1 will have to wait for T3 to finish.

- In a similar way there may exist a sequence of transactions requesting shared lock on data item A and each releases its lock a short while after it is granted., But T1 will never get the exclusive lock on data item A.
- Thus T1 will never progress and such situation is starvation
- Concurrency control manager can be designed to prevent starvation.
- To guarantee serializability, and reduce the effect of deadlock and starvation we must follow an additional protocol concerning the positioning of the lock and unlock operations in every transaction.

- The best-known protocol is two-phase locking (2PL).

Two phase locking (2PL)

- A transaction follows the two-phase locking protocol if all locking

operations precede the first unlock operation in the transaction.

Phase 1: Growing Phase

- transaction may obtain locks
- transaction may not release locks

Phase 2: Shrinking Phase

- transaction may release locks
- transaction may not obtain locks

The two-phase locking: the protocol assures serializability. It can be proved that the transactions can be serialized in the order of their lock points (i.e. the point where a transaction acquired its final lock).

- T1
- READ(B)
- B:=B-50
- WRITE(B)
- UNLOCK(B)
- LOCKX(A)
- READ(A)
- A:=A+50
- WRITE(A)
- UNLOCK(A)
- Above example is not 2 phase because unlock(b) appears before lock(a)

- T1
- READ(B)
- B:=B-50
- WRITE(B)
- LOCKX(A)
- READ(A)
- A:=A+50
- WRITE(A)
- UNLOCK(B)
- UNLOCK(A)
- Above example is 2 phase because unlocks appears after all lock operation

- Two-phase locking does not ensure freedom from deadlocks
- Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called strict two-phase locking.
- Here a transaction must hold all its exclusive locks till it commits/aborts.
- This requirement ensures that any data written by uncommitted transaction are locked in exclusive mode until the transaction commits, preventing
  any other transaction from reading the data

TIME STAMPING METHODS

- Timestamp methods for concurrency control are quite different from locking methods. No locks are involved, and therefore there can be no deadlock.
- Locking methods generally prevent conflicts by making transactions wait.
- With timestamp methods, there is no waiting: transactions involved in conflict are simply rolled back and restarted.
- Timestamp A unique identifier created by the DBMS that indicates the relative starting time of a transaction.

A concurrency control protocol that orders transactions in such a way that older transactions, transactions with smaller timestamps, get priority in the event of conflict.

DATABASE RECOVERY

- The process of restoring the database to a correct state in the event of a failure
- There are many different types of failure that can affect database processing, each of which has to be dealt with in a different manner.
- Some failures affect main memory only, while others involve non-volatile (secondary) storage.
- among the causes of failure are:

1. System crashes: due to hardware or software errors, resulting in loss of main memory;
2. Media failures: such as head crashes or unreadable media, resulting in the loss of parts of secondary storage;
3. Application software errors: such as logical errors in the program that is accessing the database, which cause one or more transactions to fail;
4. Natural physical disasters: such as fires, floods, earthquakes, or power failures;
5. Carelessness: or unintentional destruction of data or facilities by operators or users;
6. Sabotage, or intentional corruption or destruction of data, hardware, or software facilities

Whatever the cause of the failure, there are two principal effects that

we need to consider: the loss of main memory, including the database buffers, and the loss of the disk copy of the database

TRANSACTIONS AND RECOVERY

Transactions represent the basic unit of recovery in a database system.

- It is the role of the recovery manager to guarantee two of the four ACID properties of transactions, namely **atomicity and durability**, in the presence of failures.

RECOVERY TECHNIQUES AND FACILITIES

Recovery Facilities

- A DBMS should provide the following facilities to assist with recovery: a backup mechanism, which makes periodic backup copies of the database;
- The backup copy of the database can be used in the event that the database has been damaged or destroyed
- Typically, the backup is stored on offline storage, such as magnetic tape.
- logging facilities, which keep track of the current state of transactions and database changes;
- The DBMS maintains a special file called a log (or journal) that contains information about all updates to the database

A checkpoint facility, which enables updates to the database that are in progress to be made permanent;

- The point of synchronization between the database and the transaction log file. All buffers are force-written to secondary storage.
- Used to limit the amount of searching and subsequent processing that

we need to carry out on the log file

- Checkpoints are scheduled at predetermined intervals and involve the following operations:
  - Writing all log records in main memory to secondary storage;
  - writing the modified blocks in the database buffers to secondary storage;
  - writing a checkpoint record to the log file. This record contains the identifiers of all transactions that are active at the time of the checkpoint.
- A recovery manager, which allows the system to restore the database to a consistent state following a failure.

Ze End