Unit - V

TRANSACTION MANAGEMENT
TRANSACTIONS

CONTENTS

- **TRANSACTION MANAGEMENT TRANSACTIONS:** Transaction Concept, Transaction State, Implementation of Atomicity and Durability, Concurrent Executions, Serializability, Recoverability, Implementation of Isolation, Testing for Serializability.
- **Concurrency Control:** Lock based Protocols, Timestamp based protocols
- **Recovery System:** Recovery and Atomicity, Log based recovery, Shadow Paging, Recovery with concurrent Transactions, Buffer Management.

Transaction

- The transaction is a set of logically related operation. It contains a group of tasks.
- A transaction is an action or series of actions.
- It is performed by a single user to perform operations for accessing the contents of the database.

- **Example:** Suppose an employee of bank transfers Rs 800 from X's account to Y's account. This small transaction contains several low-level tasks:

Example:

- **X's Account**
- Open_Account(X)
- Old_Balance = X.balance
- New_Balance = Old_Balance - 800
- X.balance = New_Balance
- Close_Account(X)
- **Y's Account**
- Open_Account(Y)
- Old_Balance = Y.balance
- New_Balance = Old_Balance + 800
- Y.balance = New_Balance
- Close_Account(Y)

Operations of Transaction

- Following are the main operations of transaction:
  1. **Read(X):** Read operation is used to read the value of X from the database and stores it in a buffer in main memory.
  2. **Write(X):** Write operation is used to write the value back to the database from the buffer.

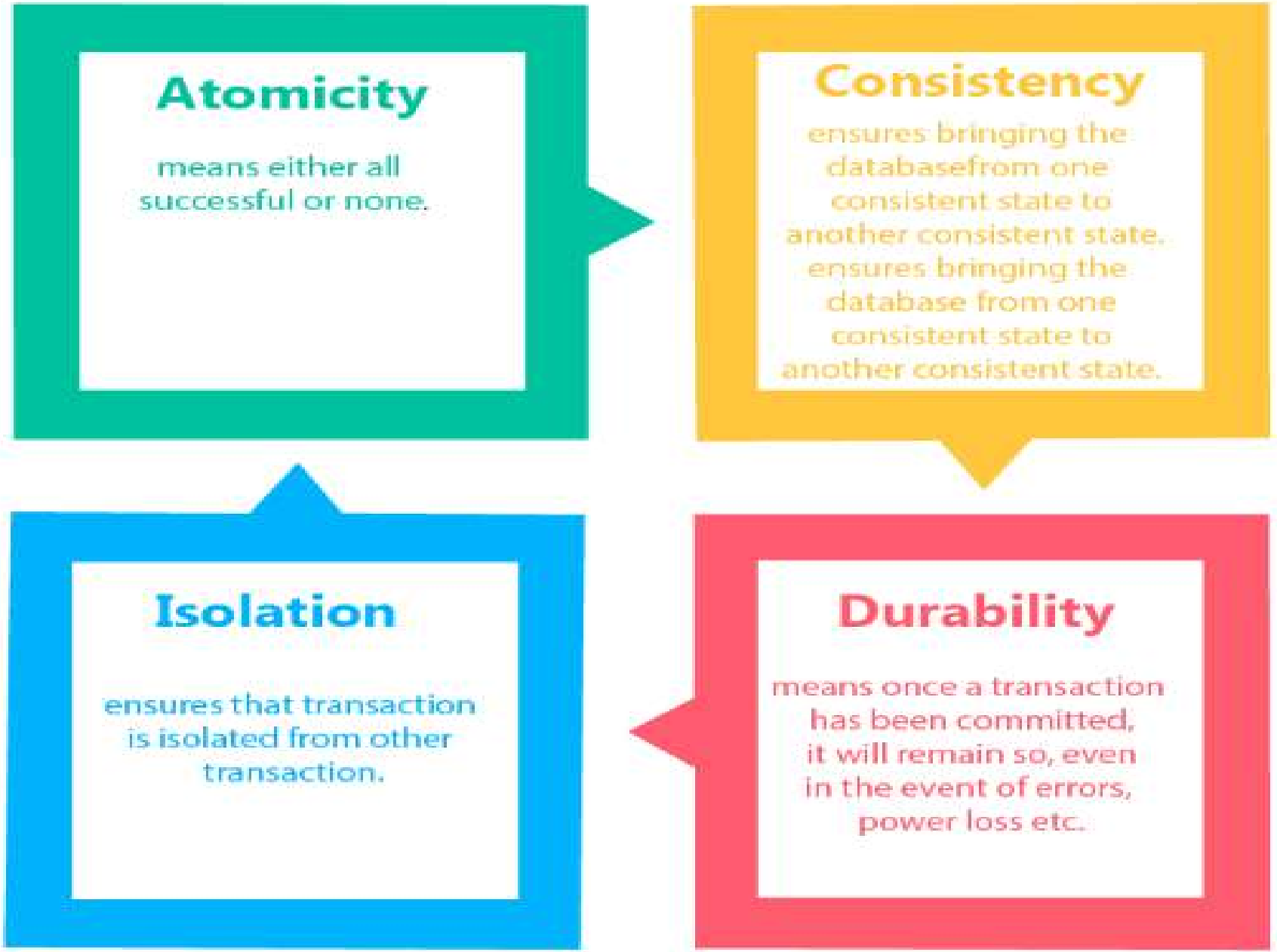
- **Example:**
  - 1. R(X);
  - 2. X = X - 500;
  - 3. W(X);

Example:

- If in the above transaction, the debit transaction fails after executing operation 2 then X's value will remain 4000 in the database which is not acceptable by the bank.
- To solve this problem, we have two important operations:
  1. **Commit:** It is used to save the work done permanently.
  2. **Rollback:** It is used to undo the work done.

Transaction property

- The transaction has the four properties. These are used to maintain consistency in a database, before and after the transaction.
- **Property of Transaction**
  1. Atomicity
  2. Consistency
  3. Isolation
  4. Durability


Atomicity
means either all successful or none.
Consistency
ensures bringing the databasefrom one consistent state to another consistent state. ensures bringing the database from one consistent state to another consistent state.
Isolation
ensures that transaction is isolated from other transaction.
Durability
means once a transaction has been committed, it will remain so, even in the event of errors, power loss etc.

1. Atomicity

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

2.Consistency

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

3.Isolation

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

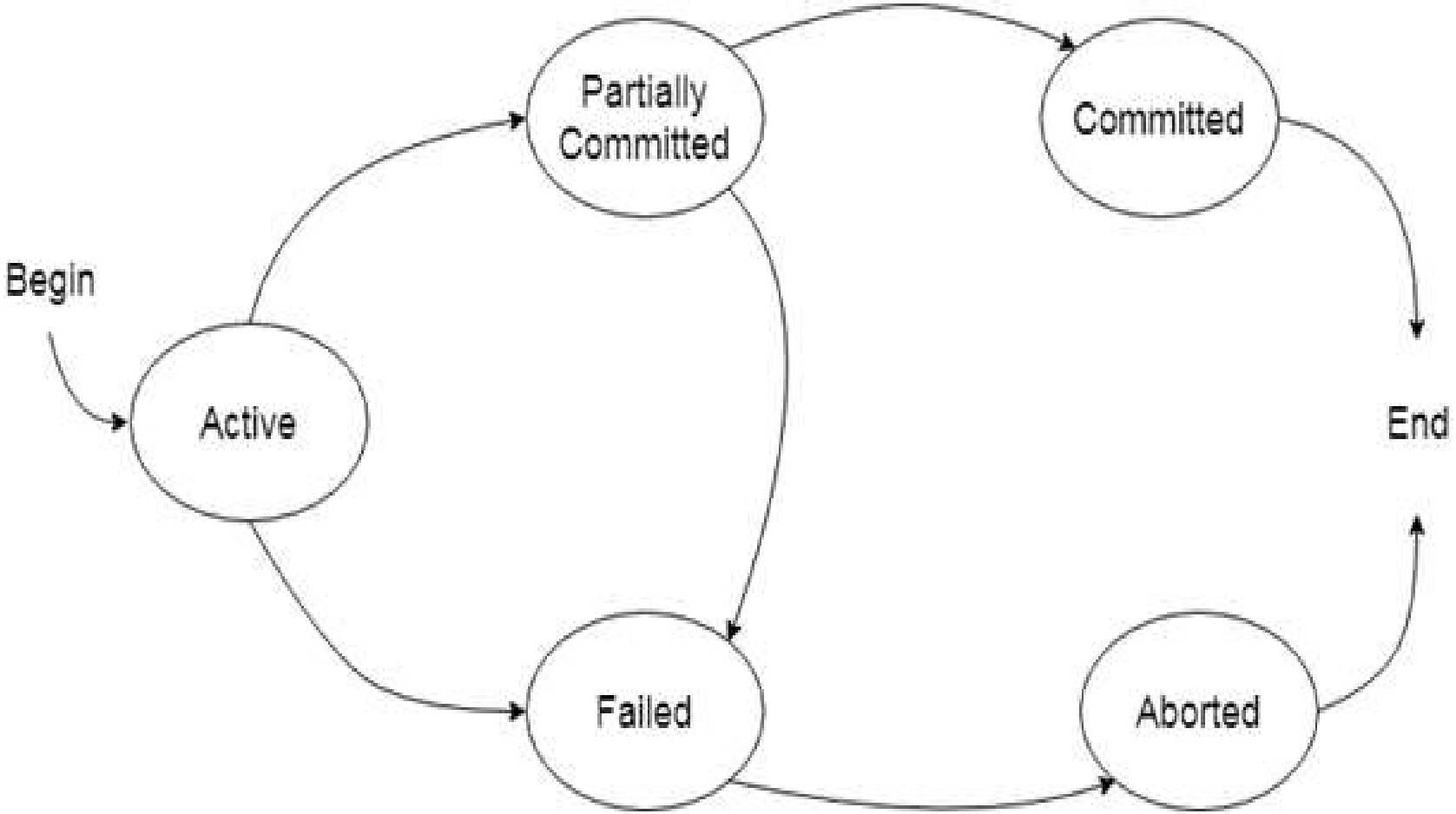
4.Durability

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

Transaction States

- **Active:** It is a initial state, the transaction stays in this state while it is executing.
- **Partially committed**: after the final statement has been executed.
- **Committed**: it's a state after successful completion.
- **Failed**: it is discovered that normal execution can no longer proceed.
- **Aborted** : it's a state when the transaction has been rolled back and database has been restored to its state prior to the state of transaction.

State Diagram

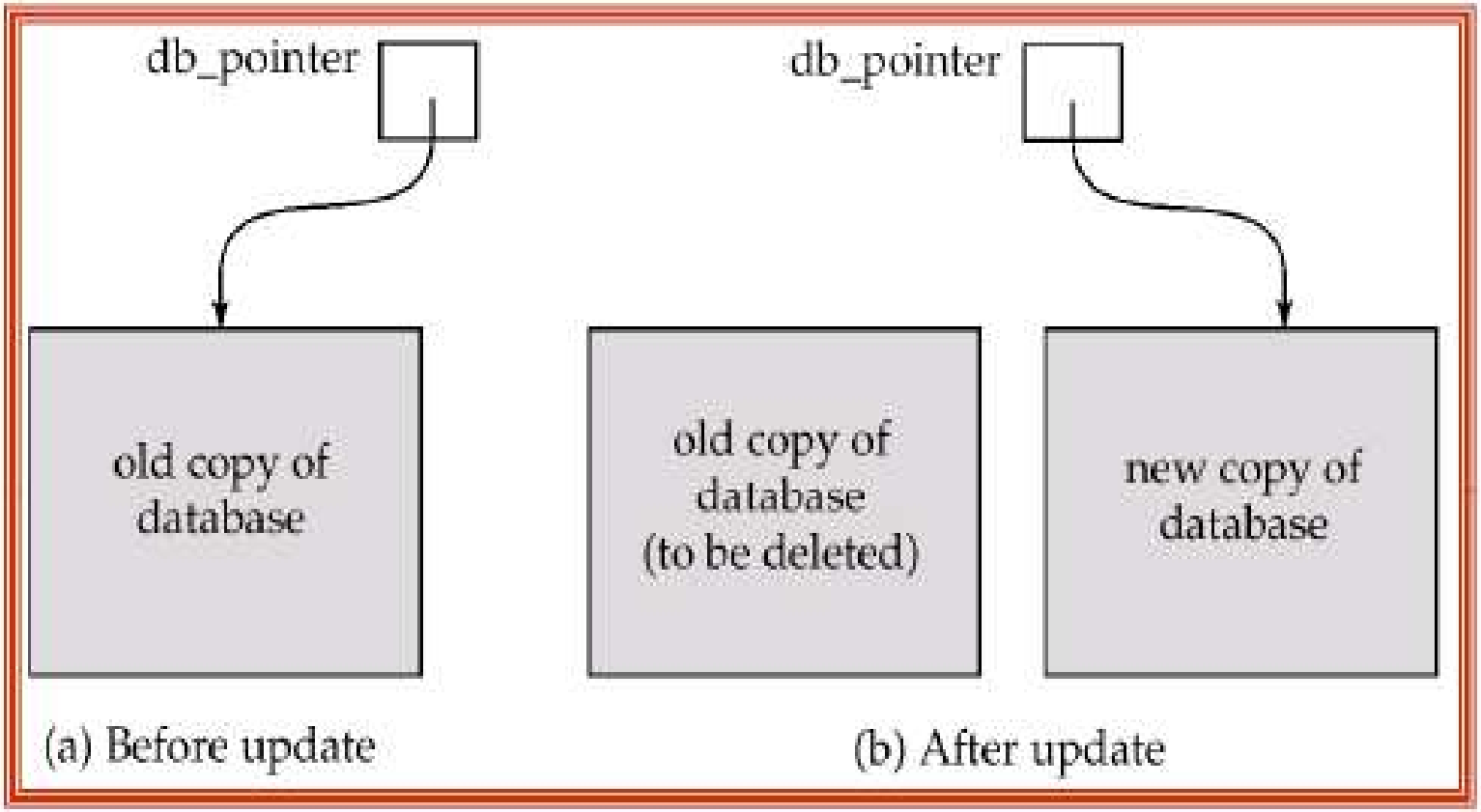


Implementation of Atomicity & Durability

- The recovery-management component of a database system implements the support for atomicity and durability.

The shadow-database scheme:

- Assume that only one transaction is active at a time.
- A pointer called db_pointer always points to the current consistent copy of the database.
- All updates are made on a shadow copy of the database, and db_pointer is made to point to the updated shadow copy only after the transaction reaches partial commit and all updated pages have been flushed to disk.
- In case transaction fails, old consistent copy pointed to by db_pointer can be used, and the shadow copy can be deleted.



- This schema is extremely inefficient for large databases
- Executing a single transaction requires copying the entire database.

Concurrent Executions

Multiple transactions are allowed to run concurrently in the system.

Advantages are:

Improve throughput:

- Increased processor and disk utilization, leading to better transaction.
- One transaction can be using the CPU while another is reading from or writing to the disk reduced

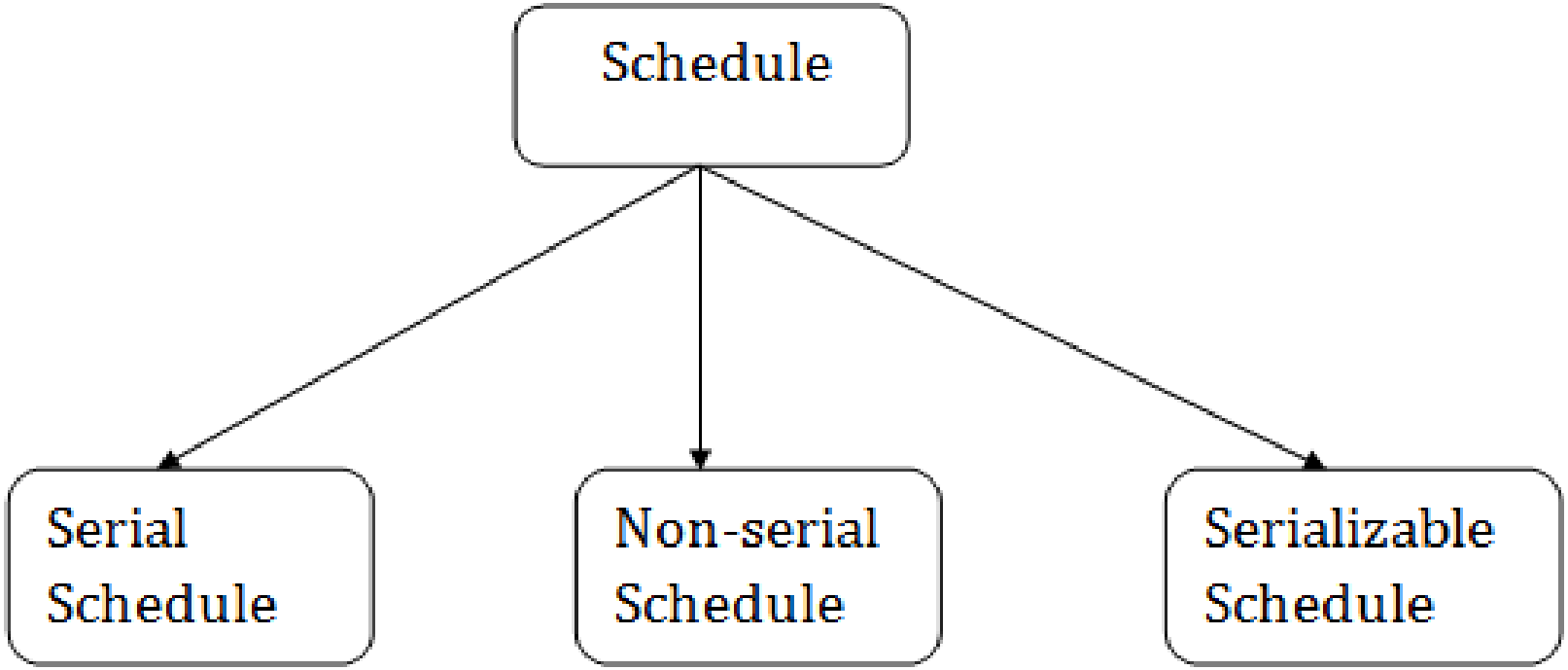
Reduced waiting time:

- Concurrent execution reduces the unpredictable delays in running a transaction which in turn reduces average response time for transactions.

Schedule

- A series of operation from one transaction to another transaction is known as schedule.
- It is used to preserve the order of the operation in each of the individual transaction.
  - A schedule that contains either an abort or commit for each transaction is called as ***complete schedule***.
  - If the action of different transactions are not interleaved then we call the schedule as ***serial schedule.***

Schedule Types:



Serial Schedule:

- The serial schedule is a type of schedule where one transaction is executed completely before starting another transaction. In the serial schedule, when the first transaction completes its cycle, then the next transaction is executed.

- **For example:** Suppose there are two transactions T1 and T2 which have some operations. If it has no interleaving of operations, then there are the following two possible outcomes:
  - Execute all the operations of T1 which was followed by all the operations of T2.
  - In the given (a) figure, Schedule A shows the serial schedule where T1 followed by T2.
  - In the given (b) figure, Schedule B shows the serial schedule where T2 followed by T1.

Non-Serial Schedule:

- If interleaving of operations is allowed, then there will be non-serial schedule.
- It contains many possible orders in which the system can execute the individual operations of the transactions.
- In the given figure (c) and (d), Schedule C and Schedule D are the non-serial schedules. It has interleaving of operations.

Serializable Schedule:

- The serializability of schedules is used to find non-serial schedules that allow the transaction to execute concurrently without interfering with one another.
- It identifies which schedules are correct when executions of the transaction have interleaving of their operations.
- A non-serial schedule will be serializable if its result is equal to the result of its transactions executed serially.

Example:

Time

T1	T2

read(A);
A := A – N;
write(A);
read(B);
B := B + N;
write(B);
	read(A);
	A := A + M;
	write(A);

Schedule A

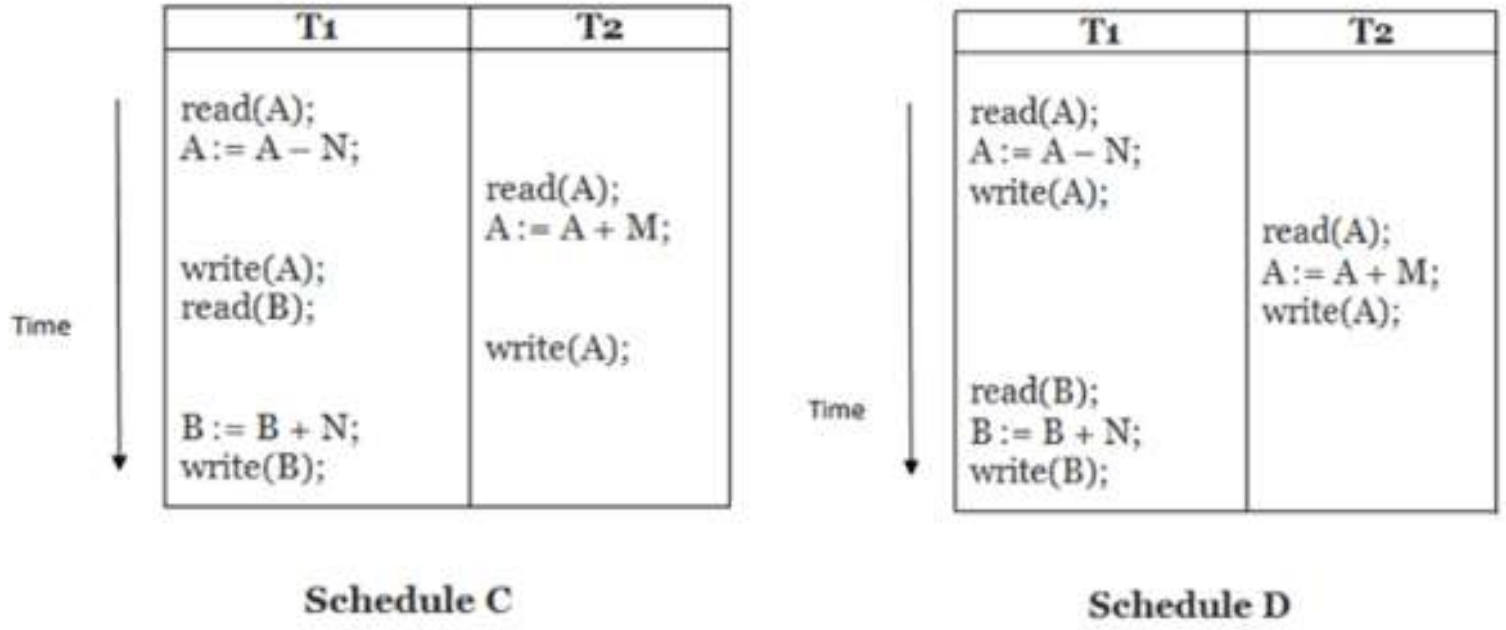
Time

T1	T2

read(A);
A:= A - N;
write(A);
read(B);
B:= B +N;
write(B);
	read(A);
	A := A + M;
	write(A);

Schedule B

Example:



- **Here,**
- Schedule A and Schedule B are serial schedule.
- Schedule C and Schedule D are Non-serial schedule.

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

Recoverability of Schedule

- Sometimes a transaction may not execute completely due to a software issue, system crash or hardware failure.
- In that case, the failed transaction has to be rollback. But some other transaction may also have used value produced by the failed transaction.
- So we also have to rollback those transactions.

T1	T1's buffer space	T2	T2's buffer space	Database
				A = 6500
Read(A);	A = 6500			A = 6500
A = A - 500;	A = 6000			A = 6500
Write(A);	A = 6000			A = 6000
		Read(A);	A = 6000	A = 6000
		A =A + 1000;	A = 7000	A = 6000
		Write(A);	A = 7000	A = 7000
		Commit;		
Failure Point				
Commit;				

Concurrency Control

Lock based Protocols

- A lock is a mechanism that tells the DBMS whether a particular data item is being used by any transaction for read/write purpose.
- Depending upon the rules we have found, we can classify the locks into two types.

Shared Lock: A transaction may acquire shared lock on a data item in order to read its content. The lock is shared in the sense that any other transaction can acquire the shared lock on that same data item for reading purpose.

Exclusive Lock: A transaction may acquire exclusive lock on a data item in order to both read/write into it. The lock is excusive in the sense that no other transaction can acquire any kind of lock (either shared or exclusive) on that same data item.

- The relationship between Shared and Exclusive Lock can be represented by the following table which is known as **Lock Matrix**.

	Shared	Exclusive
Shared	TRUE	FALSE
Exclusive	FALSE	FALSE

- In a transaction, a data item which we want to read/write should first be locked before the read/write is done.
- After the operation is over, the transaction should then unlock the data item so that other transaction can lock that same data item for their respective usage.

Granting of locks

- Lock has to be granted by concurrent control manager.
- Suppose a transaction T2 has shared-mode lock on data item.
- Other transaction T1 request an exclusive-mode lock on data item.
- In this situation T1 has to wait for T2.
- Meanwhile a transaction T3 may request a shared lock on same data item as it is compatible with T2 it can granted...but still T1 will be waiting and so on
- This situation is called as **Starvation.**

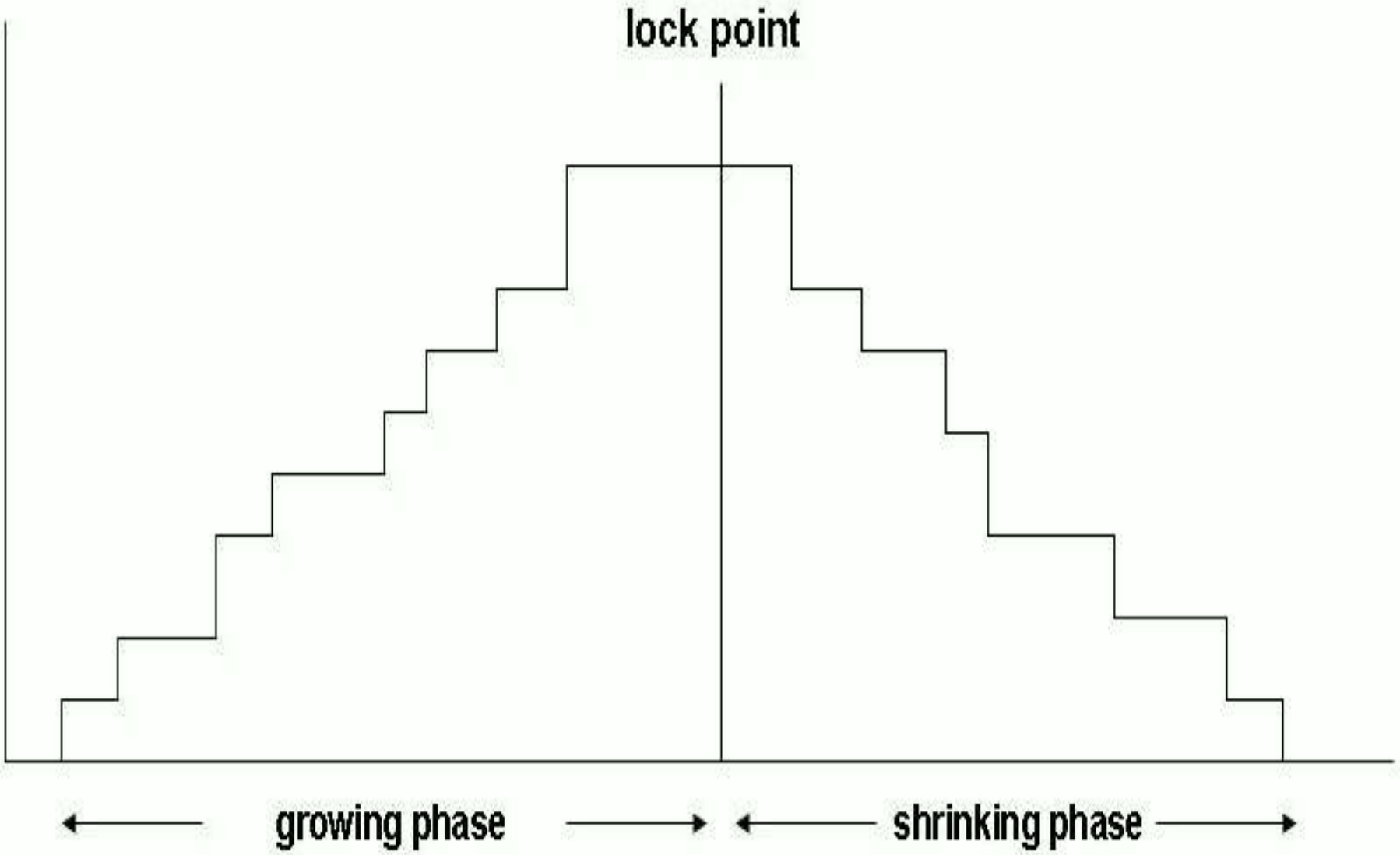
- We can avoid starvation of transaction by granting locks in the following manner.
- When a transaction Ti requested for lock on the data item Q in a mode of M. the concurrent control manager grants the lock provided that
  - There is no other transaction holding a lock on Q in a mode that conflict with M.
  - There is no other transaction that is waiting for a lock on Q and that made its lock request before Ti.
- So that a lock request will never get blocked by a lock request that is made later.

Two Phase Locking

- The Two Phase Locking Protocol defines the rules of how to acquire the locks on a data item and how to release the locks.
- The Two Phase Locking Protocol assumes that a transaction can only be in one of two phases.

Growing Phase:

- In this phase the transaction can only acquire locks, but cannot release any lock.
- The transaction enters the growing phase as soon as it acquires the first lock it wants.
- It cannot release any lock at this phase even if it has finished working with a locked data item.
- Ultimately the transaction reaches a point. This point is called **Lock Point**.


lock point
growing phase
shrinking phase

Shrinking Phase:

- After Lock Point has been reached, the transaction enters the shrinking phase.
- In this phase the transaction can only release locks, but cannot acquire any new lock.
- The transaction enters the shrinking phase as soon as it releases the first lock after crossing the Lock Point.
- Here it keeps releasing all the acquired locks.
- There are two different versions of the Two Phase Locking Protocol.

- **Strict Two Phase Locking Protocol**
- **Rigorous Two Phase Locking Protocol**

Strict Two Phase Locking Protocol

- In this protocol, a transaction may release all the shared locks after the Lock Point has been reached.
- But it cannot release any of the exclusive locks until the transaction commits.
- This protocol helps in creating cascade less schedule.

Rigorous Two Phase Locking Protocol

- A transaction is not allowed to release any lock (either shared or exclusive) until it commits.
- This means that until the transaction commits, other transaction might acquire a shared lock on a data item on which the uncommitted transaction has a shared lock.
- But cannot acquire any lock on a data item on which the uncommitted transaction has an exclusive lock

Time Stamp based Protocol

- Each transaction is issued a timestamp when it enters the system. If an old transaction Ti has time-stamp TS(Ti), a new transaction Tj is assigned time-stamp TS(Tj) such that TS(Ti) <TS(Tj).

There are 2 simple methods for implementing this scheme

1.Use a value of the system clock as the time stamp

2.Use a logical counter that is incremented after a new time stamp has been assigned.

- The protocol manages concurrent execution such that the timestamps determine the serializability order.
- In order to assure such behavior, the protocol maintains for each data Q two timestamp values:

1. **W-timestamp(Q)** is the largest time-stamp of any transaction that executed write(Q) successfully.
2. **R-timestamp(Q)** is the largest time-stamp of any transaction that executed read(Q) successfully

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

Suppose a transaction Ti issues a read(Q)

1. 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.
2. If TS(Ti)>= W-timestamp(Q), then the read operation is executed.

Suppose that transaction Ti issues write(Q).

1. If TS(Ti) < R-timestamp(Q), In such a case Ti has lost its relevance and will be rolled back.
2. If TS(Ti) < W-timestamp(Q), Ti has lost its relevance and will be rolled back.
3. Otherwise, the write operation is executed

Validation based Protocols

- Execution of transaction Ti is done in three phases.

1. **Read phase**:
2. It reads the value of various data items and store them in variables local to Ti
3. Transaction Ti writes only to temporary local variables, with out updates on actual data base.

2. **Validation phase**:

Transaction Ti performs a “validation test'‘ to determine if local variables can be written without violating serializability.

3. **Write phase:**

If Ti is validated, the updates are applied to the database; otherwise, Ti is rolled back.

- The three phases of concurrently executing transactions can be interleaved.
- But each transaction must go through the three phases in that order.

Each transaction Ti has 3 timestamps

- **Start(Ti)** : the time when Ti started its execution.
- **Validation(Ti):** the time when Ti entered its validation phase.
- **Finish(Ti)** : the time when Ti finished its write phase.

The validation test for transaction Tj requires that for all transaction Ti with TS(Ti)<TS(Tj) one of the following conditions must hold.

1. Finish(Ti) < start(Tj)
2. Start(Tj) < finish(Ti) < validation(Tj) i.e the set of data items written by Ti does not intersect with the set of data items read by Tj. And Ti completes its write phase before Tj starts its validation.

Recovery and Atomicity

- Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state.
- Consider transaction Ti that transfers $50 from account A to account B; goal is either to perform all database modifications made by Ti or none at all.
- Several output operations may be required for Ti (to output A and B). A failure may occur after one of these modifications have been made but before all of them are made.
- To achieve our goal of atomicity, we must first output information describing the modifications to a stable storage, with out modifying the data base itself.

Log based Recovery

- Most widely used structure for recording data bases modifications in the log records, recording all the update activities in the data base.
  - <Ti starts> transaction Ti has started.
  - <Ti, Xj, V1, V2>. Transaction Ti has performed 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.
- When ever a transaction perform a write, it is essential that the log record for that write be created before the data base is modified.
- Once the log record exists we can output the modification to the data base if that is desirable.

- During recovery after a crash, a transaction needs to be redone if and only if both <Ti start> and<Ti commit> are there in the log.
- Redoing a transaction Ti (redoTi) sets the value of all data items updated by the transaction to the new values.

example transactions T_0 and T_1 (T_0 executes before T_1):

T_0	T_1
read (A)	**read** (C)
A: - A - 50	C:- C- 100
Write (A)	**write** (C)
read (B)	
B:- B + 50	
write (B)	

Below we show the log as it appears at three instances of time.

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 950>	<T_0, A, 950>	<T_0, A, 950>
<T_0, B, 2050>	<T_0, B, 2050>	<T_0, B, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 600>	<T_1, C, 600>
		<T_1 commit>

If log on stable storage at time of crash is as in case:

(a) No redo actions need to be taken

(b) redo(T_0) must be performed since <T_0 **commit**> is presen

(c) **redo**(T_0) must be performed followed by redo(T_1) since <T_0 **commit**> and <T_i commit> are present

Recovery procedure has two operations.

- **undo(Ti)** restores the value of all data items updated by Ti to their old values, going backwards from the last log record for Ti
- **redo(Ti)** sets the value of all data items updated by Ti to the new values, going forward from the first log record for Ti

When recovering after failure:

- Transaction Ti needs to be undone if the log contains the record <Ti start>, but does not contain the record <Ti commit>.
- Transaction Ti needs to be redone if the log contains both the record <Ti start> and the record <Ti commit>.

Below we show the log as it appears at three instances of time.

(a)	(b)	(c)
$<T_0$ start>	$<T_0$ start>	$<T_0$ start>
$<T_0, A, 1000, 950>$	$<T_0, A, 1000, 950>$	$<T_0, A, 1000, 950>$
$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$
	$<T_0$ commit>	$<T_0$ commit>
	$<T_1$ start>	$<T_1$ start>
	$<T_1, C, 700, 600>$	$<T_1, C, 700, 600>$
		$<T_1$ commit>

Recovery actions in each case above are:

(a) undo (T_0): B is restored to 2000 and A to 1000.

(b) undo (T_1) and redo (T_0): C is restored to 700, and then *A* and *B* are set to 950 and 2050 respectively.

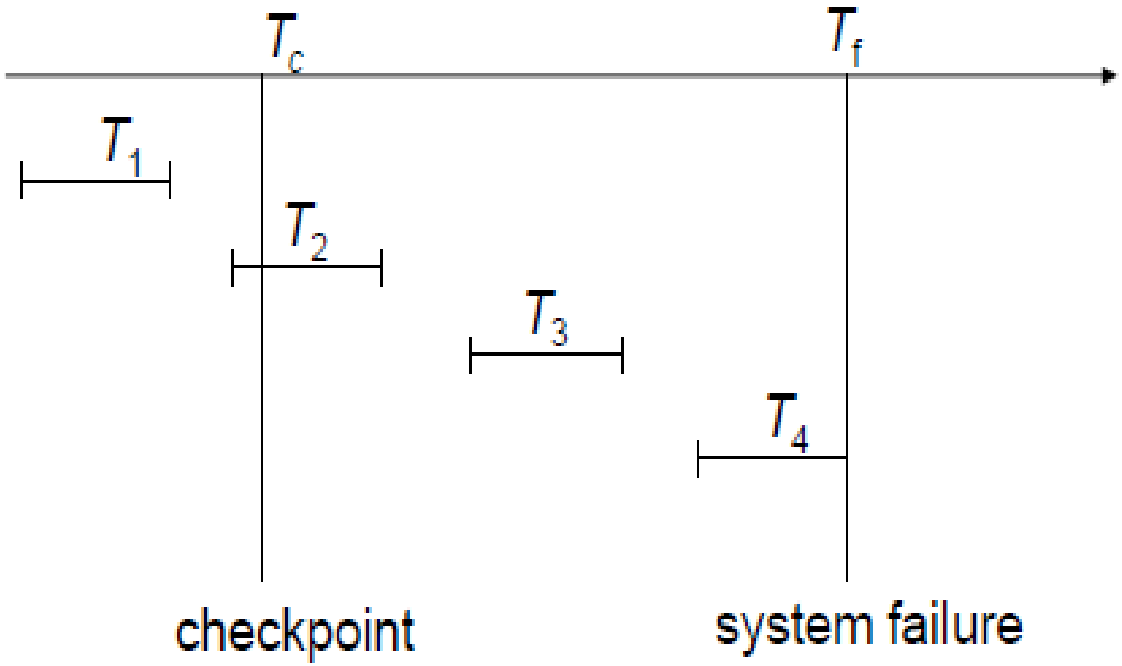
(c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then *C* is set to 600

Check points

- When a system failure occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone.
- Rather than reprocessing the entire log, which is time-consuming and much of it unnecessary, we can use *checkpoints*:

Streamline recovery procedure by periodically performing check pointing

- Output all log records currently residing in main memory onto stable storage.
- Output all modified buffer blocks to the disk.
- Write a log record < checkpoint> onto stable storage.



- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
- T_4 undone

Transaction rollback

- The system scans the log backward
- For every log record of the form <Ti,X,V1,V2> the system restores the data item X to its old value V1.
- Scanning of log terminates when the log record <Ti, start> is found.
- Scanning the log back ward is important because

  <Ti,A,10,20>

  <Ti,A,20,30>
- Scanning of log backward sets A to 10 which is correct.
- Scanning of log forward sets A to 20 which is wrong.

Restart recovery

The system construct the two lists as follows

Initially ,both records will be empty.

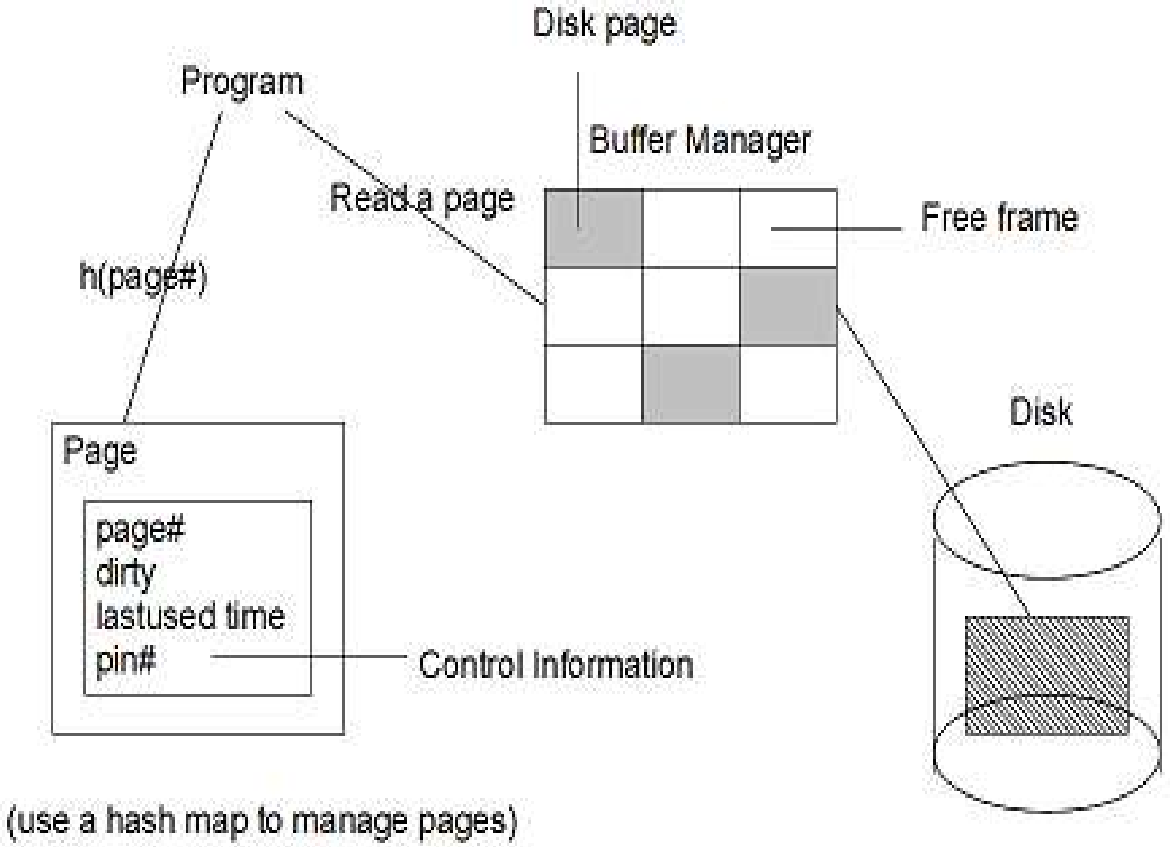
- For each record found of the form <Ti, commit> it add Ti to redo list.
- For each record found of the form < Ti start>,if Ti is not in redo list ,then it adds Ti to undo-list
- Once we completed with list construction the recovery proceeds as follows

1. Performs undo list
2. The system locates the most recent <checkpoint, L> record on log
3. Performs redo list.

Buffer Management

Log record buffering:

- Log records are buffered in main memory, instead of being output directly to stable storage.
- Log records are output to stable storage when a block of log records in the buffer is full, or a log force operation is executed.
- Log force is performed to commit a transaction by forcing all its log records (including the commit record) to stable storage.
- Several log records can thus be output using a single output operation, reducing the I/O cost.



(use a hash map to manage pages)

The rules below must be followed if log records are buffered:

- Log records are output to stable storage in the order in which they are created.
- Transaction Ti enters the commit state only when the log record <Ti commit> has been output to stable storage.
- Before a block of data in main memory is output to the database, all log records pertaining to data in that block must have been output to stable storage.

This rule is called the write-ahead logging or WAL

Database buffering

1. Database maintains an in-memory buffer of data blocks
   - When a new block is needed, if buffer is full an existing block needs to be removed from buffer
   - If the block chosen for removal has been updated, it must be output to disk
2. As a result of the write-ahead logging rule, if a block with uncommitted updates is output to disk, log records with undo information for the updates are output to the log on stable storage first.

3. No updates should be in progress on a block when it is output to disk. Can be ensured as follows.

- Before writing a data item, transaction acquires exclusive lock on block containing the data item
- Lock can be released once the write is completed.
- Such locks held for short duration are called latches.
- Before a block is output to disk, the system acquires an exclusive latch on the block ensures no update can be in progress on the block

Role of Buffer Manager

- Buffer Manager responds to the request for main memory access to disk blocks. Below picture depicts it.

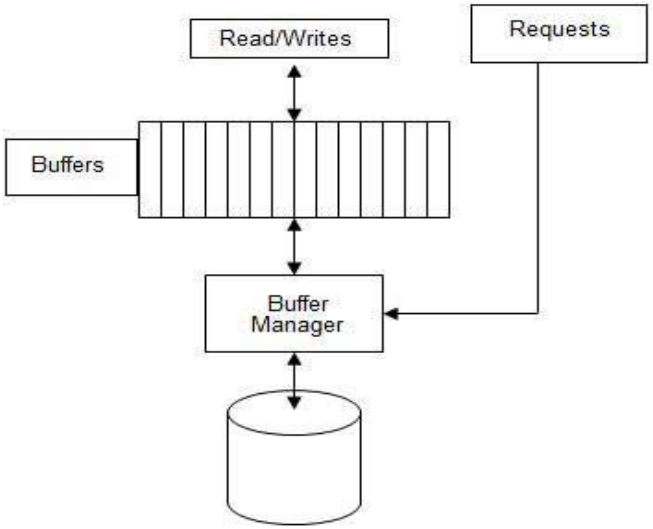


Figure 1

End of Unit 5