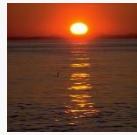
RECOVERY

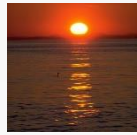


Introduction

- A computer system, like any other device, is subject to failure from a variety of causes: disk crash, power outage, software error, a fire in the machine room, even sabotage.

- In any failure, information may be lost. Therefore, the database system must take actions in advance to ensure that the atomicity and durability properties of transactions.

- An integral part of a database system is a recovery scheme that can restore the database to the consistent state that existed before the failure.



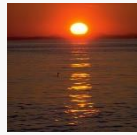
Failure Classification

- There are various types of failure that may occur in a system and each of which needs to be dealt with in a different manner.

Transaction failure:

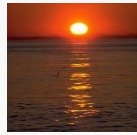
Logical errors: transaction cannot complete due to some internal error condition, bad input, no data found, overflow, or limit exceed.

System errors: the database system must terminate an active transaction due to an error condition (e.g., deadlock)



Failure Classification

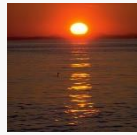
- **System crash:** a power failure or other hardware or software failure causes the system to crash (halt).
  - **Fail-stop assumption**: non-volatile storage contents are assumed to not be corrupted by system crash.
    - Database systems have numerous integrity checks to prevent corruption of disk data.
- **Disk failure**: a head crash or similar disk failure destroys all or part of disk storage
  - Destruction is assumed to be detectable: disk drives use checksums to detect failures
  - Multiple copies or archival tapes are solutions



Recovery Algorithms

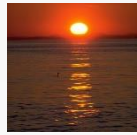
- Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures

- **Recovery algorithms have two parts**
  1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures
  2. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability



Recovery and Atomicity

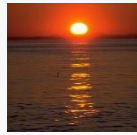
- Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state
- Consider transaction T_i that transfers \$50 from account *A* to account *B*
  - Goal is either to perform all database modifications made by T_i or none at all.
- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
- We will study two approaches:
  - **log-based recovery**
  - **shadow-paging**
- We assume (initially) that transactions run serially, that is, one after the other.



EXAMPLE

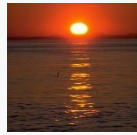
Consider again our simplified banking system and a transaction Ti that transfers $50 from account A to account B, with initial values of A and B being $1000 and $2000, respectively. Suppose that a system crash has occurred during the execution of Ti, after output (BA) has taken place, but before output (BB) was executed, where BA and BB denote the buffer blocks on which A and B reside. Since the memory contents were lost, we do not know the fate of the transaction.

When the system restarts, the value of A would be $950, while that of B would be $2000, which is clearly inconsistent with the atomicity requirement for transaction Ti.



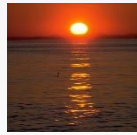
Recovery Techniques

- Log Based Crash Recovery
  - Deferred update
    - Complete -> update
  - Immediate update
    - Change -> update
  - Checkpoints
- Shadow Paging



Log-Based Recovery

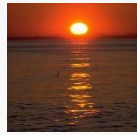
- A **log** is kept on stable storage.
  - The log is a sequence of **log records**, and maintains a record of update activities on the database.
  - An update log record has these fields:
    - **Transaction identifier**, which is the unique identifier of the transaction that performed the write operation.
    - **Data-item identifier,** which is the unique identifier of the data item written.
    - **Old value**, which is the value of the data item prior to the write.
    - **New value**, which is the value that the data item will have after the write.



Example

<T1 Start>
<T1,X,10,15>
<T1 Commit>
<T1 Abort>

Transaction 1(identifier)
X is data item whose old value is 10 and new value is 15
If transaction committed then the value of X will be 15
If abort then T1 will be rollback and value of X will be 10 again



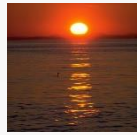
Write-Ahead Log strategy

Before updation in database Log is written for every transaction.

Operations in recovery procedure

1. **UNDO(Ti) ,means restore the old value.**
2. **REDO(Ti),Update by new value.**

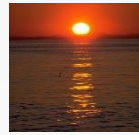
Tn is not allowed to update the physical database until undo portion of log is written.



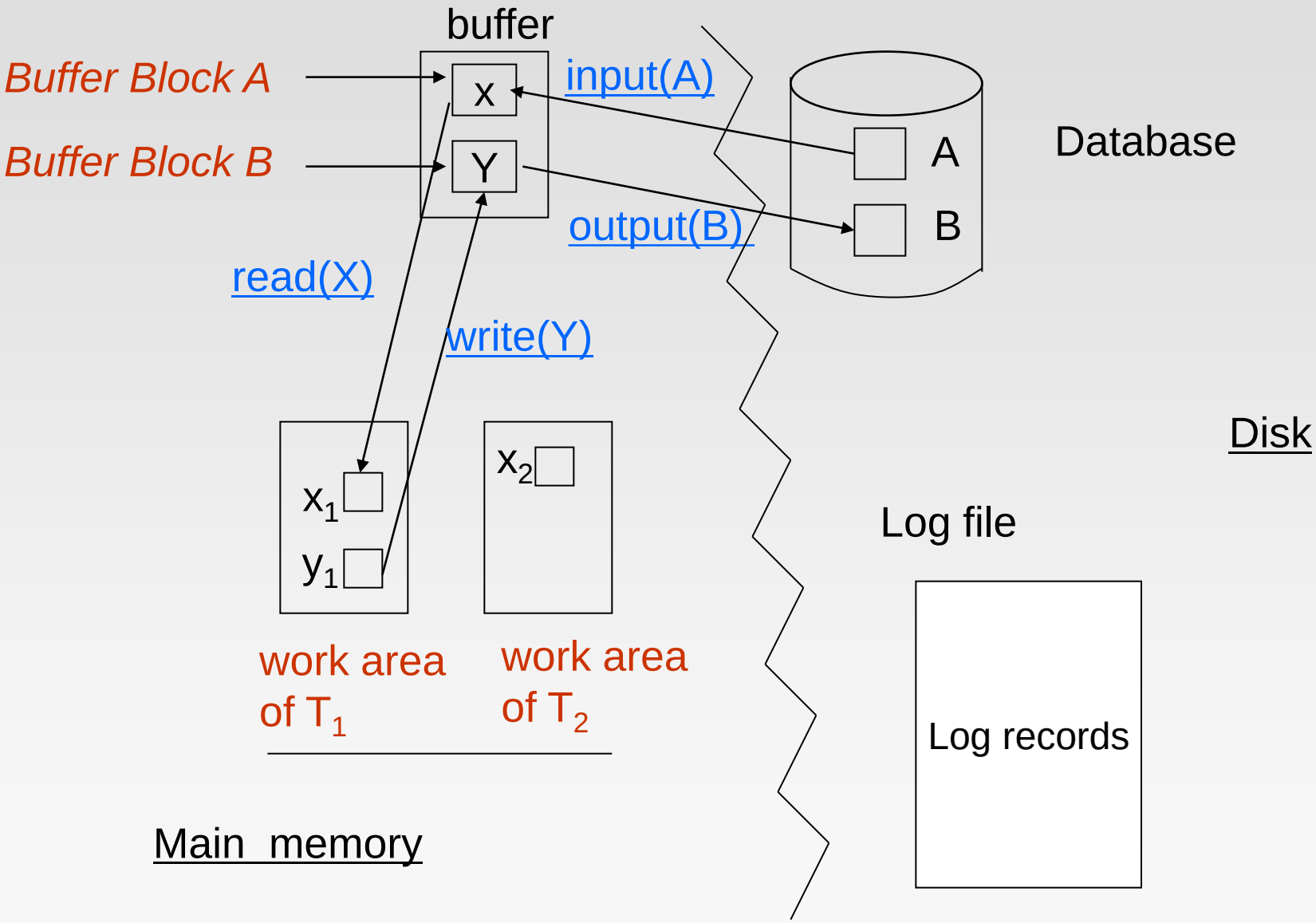
Log-Based Recovery

- **Log Records**
  - When transaction T_i starts, it registers itself by writing a $<T_i$ **start**$>$ log record
  - *Before* T_i executes **write**(X), a log record $<T_i, X, V_1, V_2>$ is written, where V_1 is the value of X before the write, and V_2 is the value to be written to X.
    - Log record notes that T_i has performed a write on data item X.
    - X had value V_1 before the write, and will have value V_2 after the write.
  - When T_i finishes it last statement, the log record $<T_i$ **commit**$>$ is written.
  - <Ti, Abort>
- We assume for now that log records are written directly to stable storage (that is, they are not buffered)
- Two approaches using logs
  - Deferred database modification
  - Immediate database modification

At the moment, Assume serial execution of Transactions T0, T1, T3…



Example of Data Access



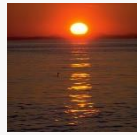
Transaction recovery Approaches

Deferred Database modification(No undo/Redo Algorithm)

- Transaction doesn’t perform immediate updation on the database.
- Only transaction log is updated.
- Database will physically update after the transaction reaches at its commit point

Immediate Database modification(Undo/No redo Algorithm)

- Database is immediately updated by transaction operations during the execution of transaction even before reaches at commit point.
- In case of transaction abort before reaches at commit point, a rollback or undo operation needs to be done to restore the database to its consistent state.



Recovery based on deferred update

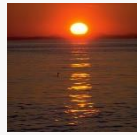
This technique postpone any actual update to the database until the transaction complete and reached check point.

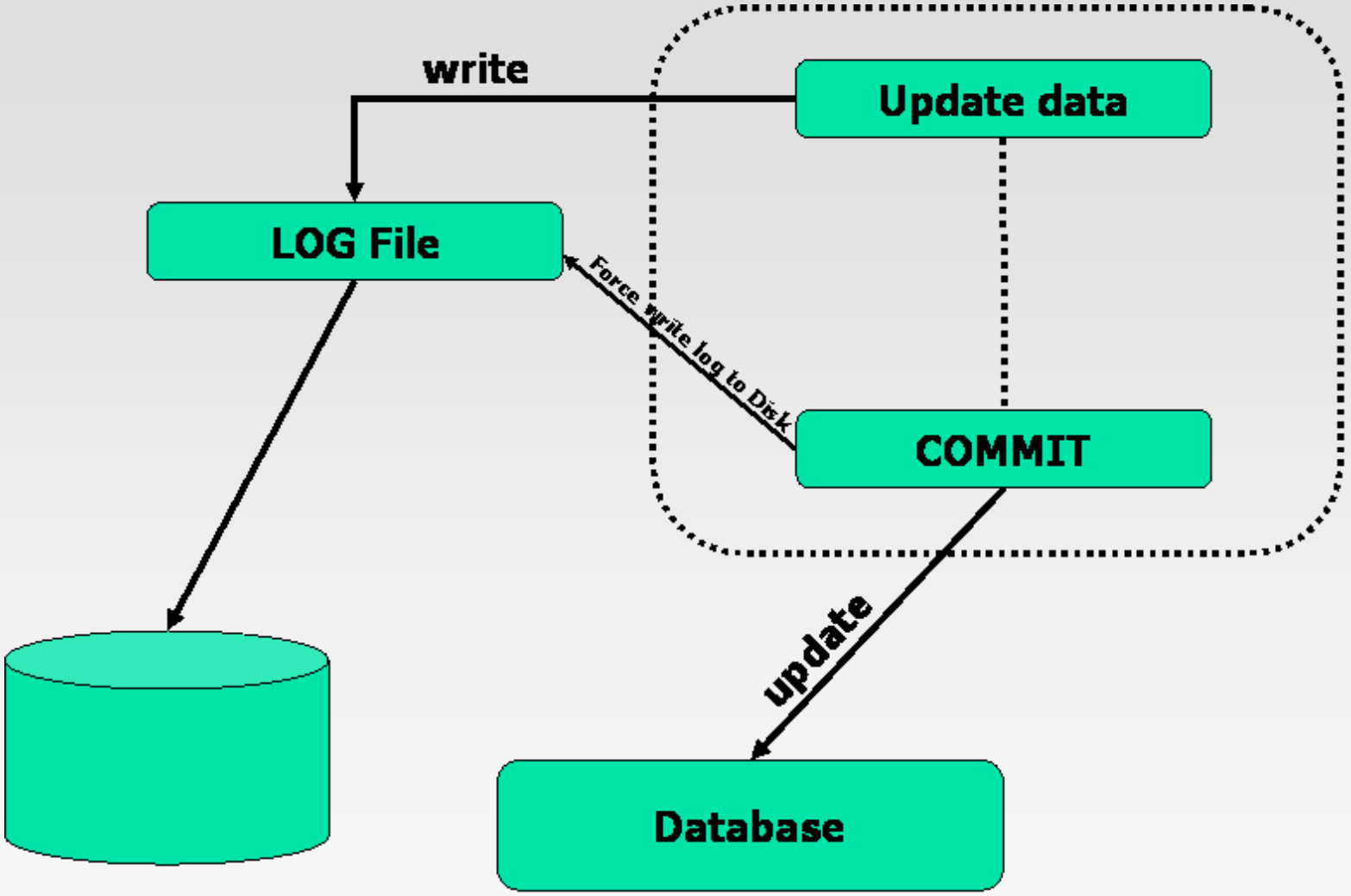
During transaction execute

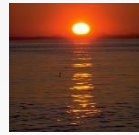
- Updates are recorded in log file and in cache buffer.
- After transaction reaches it commit point and the log file is forced to write to disk, the update are record to database.

Fail before commit,

- no need undo.

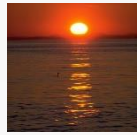



write
Update data
LOG File
Force write log to Disk
COMMIT
update
Database



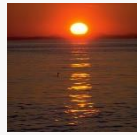
Deferred Database Modification

- The **deferred database modification** scheme records all modifications to the log, but defers all the **write**s to after partial commit.
- Assume that transactions execute serially
- Transaction starts by writing $<T_i$ ***start***$>$ record to log.
- A **write**(X) operation results in a log record $<T_i, X, V>$ being written
  - where V is the new value for X
  - Note: old value is not needed for this scheme
- Deferred Update Steps
  - "write" operation is not performed on X at this time, but is deferred.
  - When T_i partially commits, $<T_i$ **commit**$>$ is written to the log
  - Finally, the log records are read and used to actually execute the previously deferred writes.



Deferred Database Modification (Cont.)

- REDO only scheme
  - During recovery after a crash, a transaction needs to be redone if and only if both $<T_i$ **start**> and $<T_i$ **commit**> are there in the log.
  - Redoing a transaction T_i (**redo** T_i) sets the value of all data items updated by the transaction to the new values.
- Crashes can occur while
  - the transaction is executing the original updates, or
  - while recovery action is being taken



Deferred Database Modification: State of Log and Database

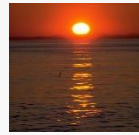
- Example transactions T_0 and T_1 (T_0 executes before T_1):

T_0: **read** (A)
$A = A - 50$
Write (A)
read (B)
$B = B + 50$
write (B)

T_1 : **read** (C)
$C = C - 100$
write (C)

State of the Log and Database corresponding to T_0 and T_1

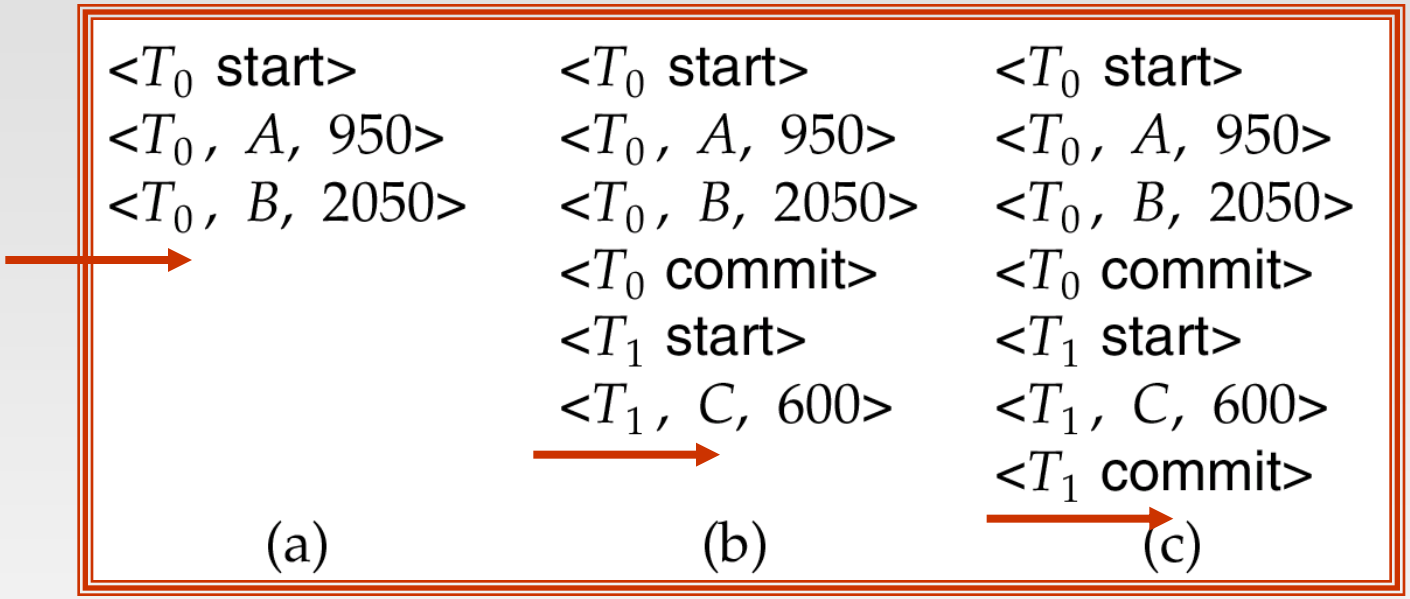
Log	Database
<T_0 start>	
<T_0, A, 950>	
<T_0, B, 2050>	
<T_0 commit>	
	$A = 950$
	$B = 2050$
<T_1 start>	
<T_1, C, 600>	
<T_1 commit>	
	$C = 600$



Deferred Database Modification (Cont.)

T0, T1, T2 …..

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



- 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 present
  - (c) { **redo**(T_0) ; **redo(T_1)**} since <T_0 **commit**> and **<** T_1 ***commit>*** are present

Example

T1	T2	T3	T4
read(A)	read(B)	read(A)	read(B)
Read(D)	write(B)	write(A)	write(B)
Write(D)	read_item(D)	read(C)	read(A)
	write(D)	write(C)	write(A)

LOG

[Start_transaction,T1]
[Write_item,T1,D,20]
[Commit,T1]
[Checkpoint]
[start_transaction,T4]
[Write_item,T4,B,15]
[Write_item,T4,A,20]
[commit,t4]
[Start_transaction,T2]
[write_item,T2,B,12]
[Start_transaction,T3]
[write_item,T3,A,30]
[write_item,T2,D,25]

System crash

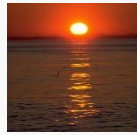
RECOVERY

- **Ignore**
  - T2,T3
- **Redone**
  - T4

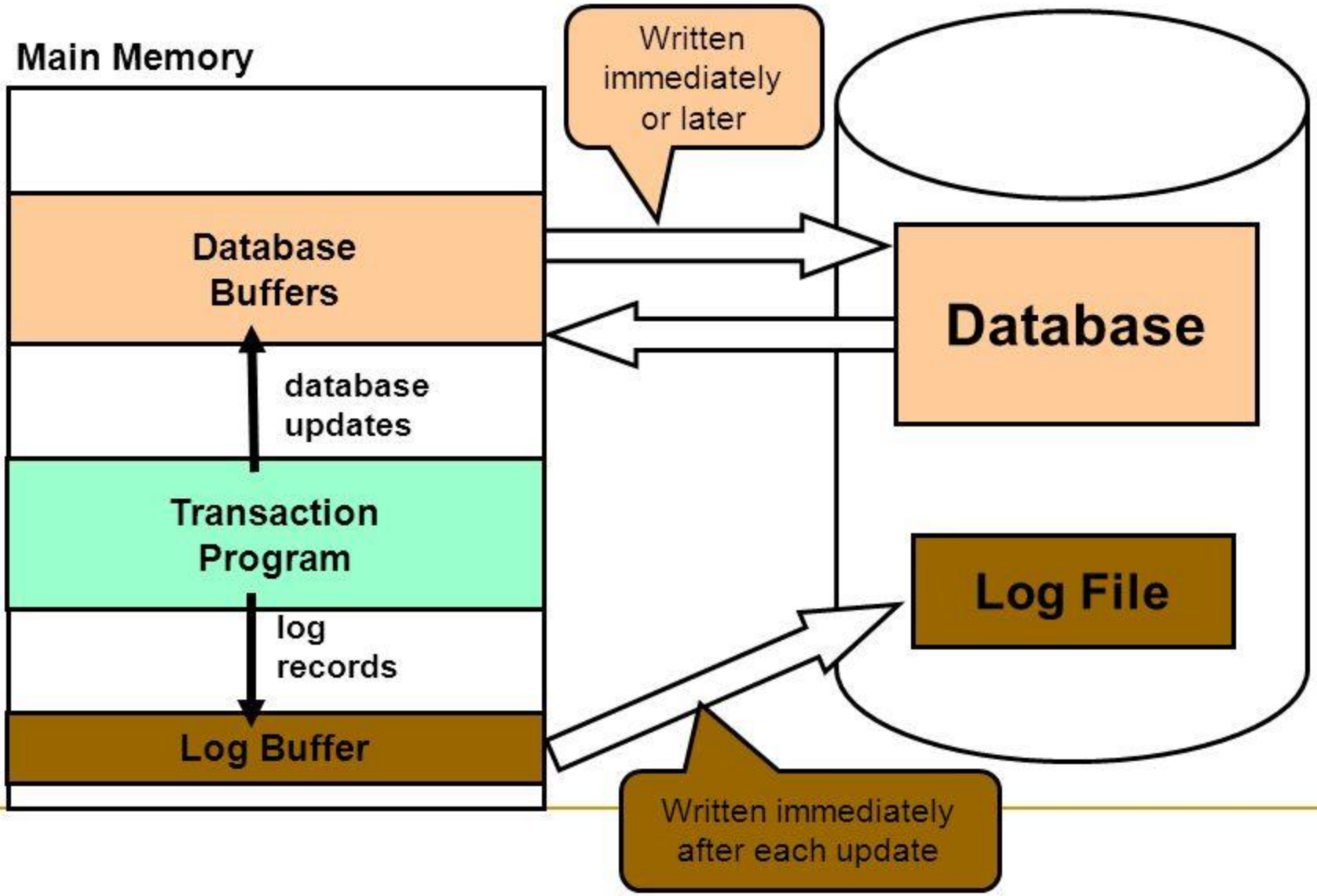
because its commit point
is after the last system checkpoint

Immediate Database Modification

- The **immediate database modification** scheme allows database updates of an uncommitted transaction to be made as the writes are issued
  - Since undoing may be needed, update logs must have both old value and new value
- Update log record must be written *before* database item is written
  - We assume that the log record is output directly to stable storage
  - Before execution of an **output**(*B*) operation for a data block B, all log records corresponding to items *B* must be flushed to stable storage
- Output of updated blocks can take place at any time before or after transaction commit
- Order in which blocks are output can be different from the order in which they are written in the buffer.



Immediate Update Layout



Immediate Database Modification Example

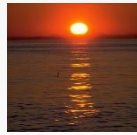
- Example transactions T_0 and T_1 (T_0 executes before T_1):

  T_0: **read** (A)
  $A = A - 50$
  Write (A)
  read (B)
  $B = B + 50$
  write (B)

  T_1 : **read** (C)
  C = C -100
  write (C)

Log	Write	Output
<T_0 **start**>		
<T_0, A, 1000, 950>		
<T_0, B, 2000, 2050>		
	A = 950	
	B = 2050	
<T_0 **commit**>		
<T_1 **start**>		
<T_1, C, 700, 600>		
	C = 600	
		B_B, B_C
<T_1 **commit**>		
		B_A

- Note: B_X denotes block containing X.



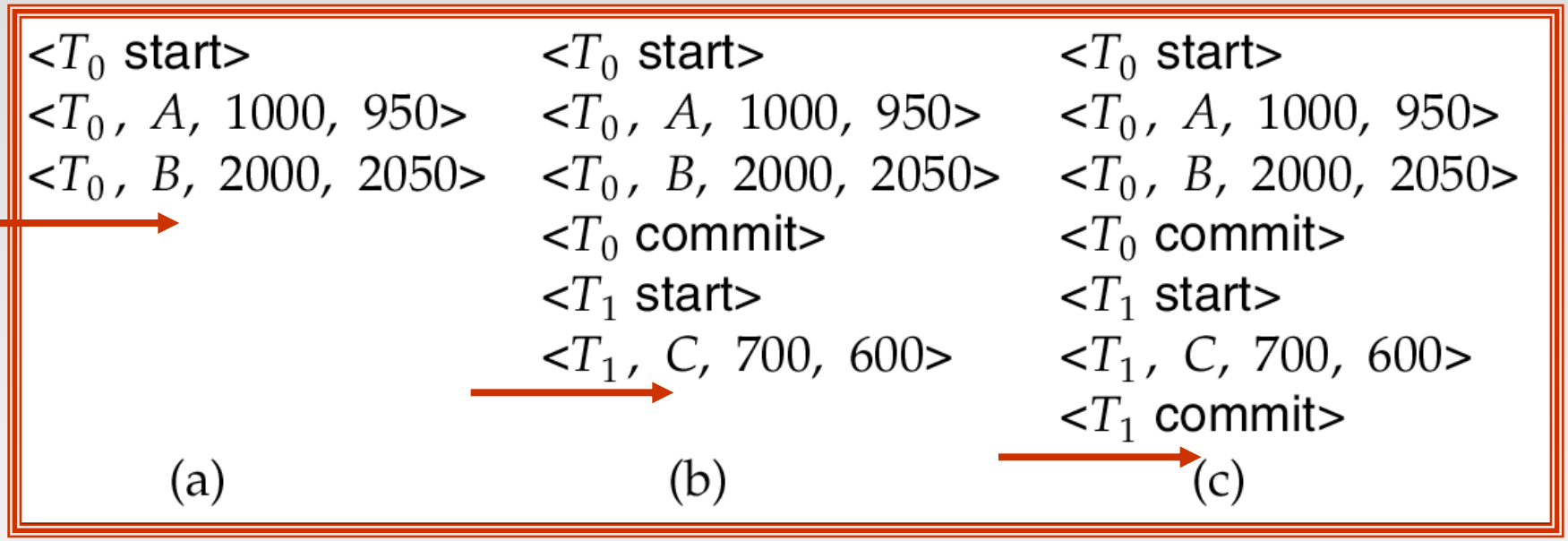
Immediate Database Modification (Cont.)

- Recovery procedure has two operations instead of one:
  - **undo**(T_i) restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i
  - **redo**(T_i) sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i
- Both operations must be **idempotent**
  - **undo**(T_i) = undo (**undo**(T_i)) = undo (undo (**undo**(T_i)))…….
  - That is, even if the operation is executed multiple times the effect is the same as if it is executed once
    - Needed since operations may get re-executed during recovery
- When recovering after failure:
  - Transaction T_i needs to be undone if the log contains the record <T_i **start**>, but does not contain the record <T_i **commit**>.
  - Transaction T_i needs to be redone if the log contains both the record <T_i **start**> and the record <T_i **commit**>.
- Undo operations are performed first, then redo operations.

Immediate DB Modification Recovery Example

Assume T0, T1, T2….

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



Recovery actions in each case above are:

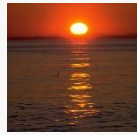
(a) undo (T_0): B is restored to 2000 and A to 1000.

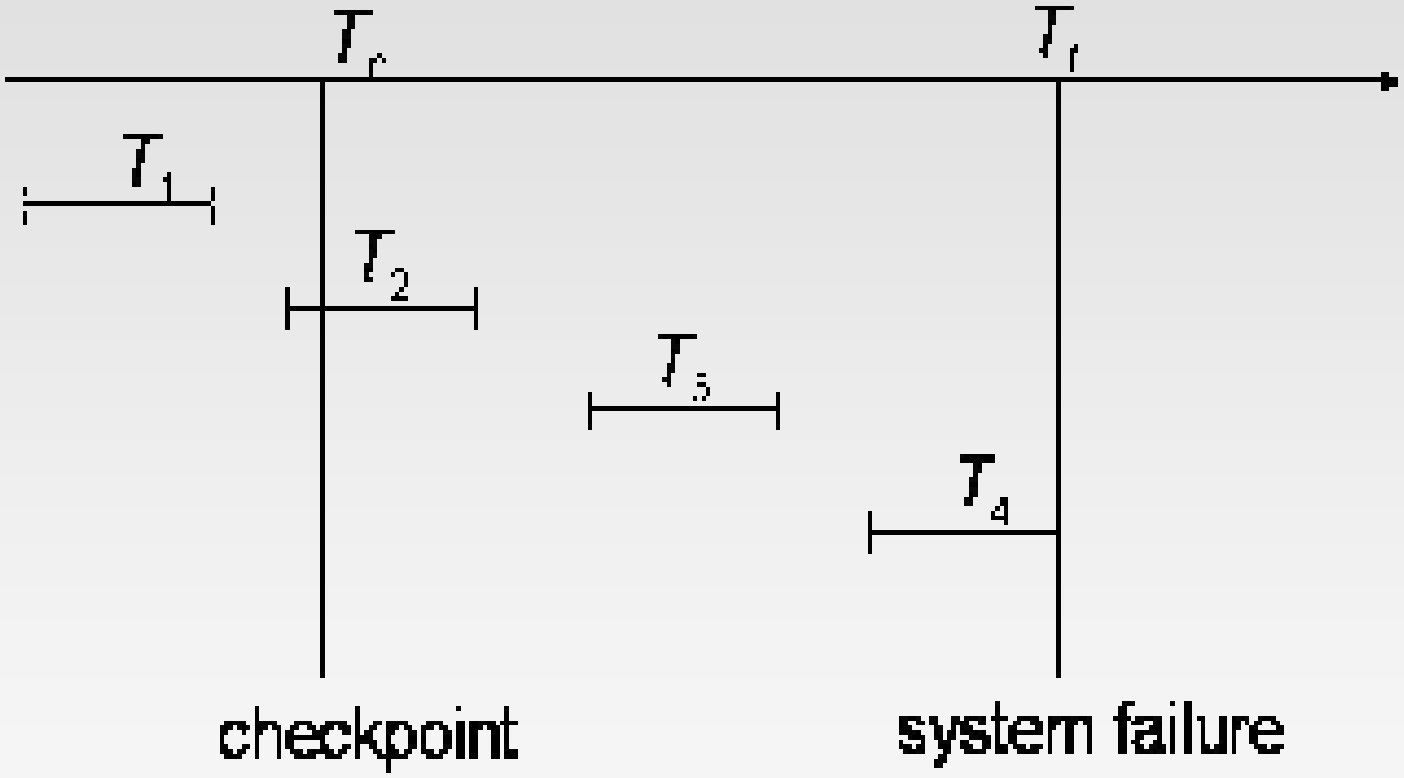
(b) undo (T_1) and redo (T_0): C is restored to 700, and then *A* and *B* are set to 950 and 2050 respectively.

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

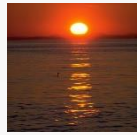
Checkpoint

- The checkpoint is a type of mechanism where all the previous logs are removed from the system and permanently stored in the storage disk.
- The checkpoint is like a bookmark. While the execution of the transaction, such checkpoints are marked, and the transaction is executed then using the steps of the transaction, the log files will be created.
- When it reaches to the checkpoint, then the transaction will be updated into the database, and till that point, the entire log file will be removed from the file. Then the log file is updated with the new step of transaction till next checkpoint and so on.
- The checkpoint is used to declare a point before which the DBMS was in the consistent state, and all transactions were committed.



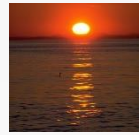

T_c
T_f
T_1
T_2
T_3
T_4
checkpoint
system failure

- T1 can be ignored (updates already output to disk due to checkpoint)
- T2 and T3 redone.
- T4 undone

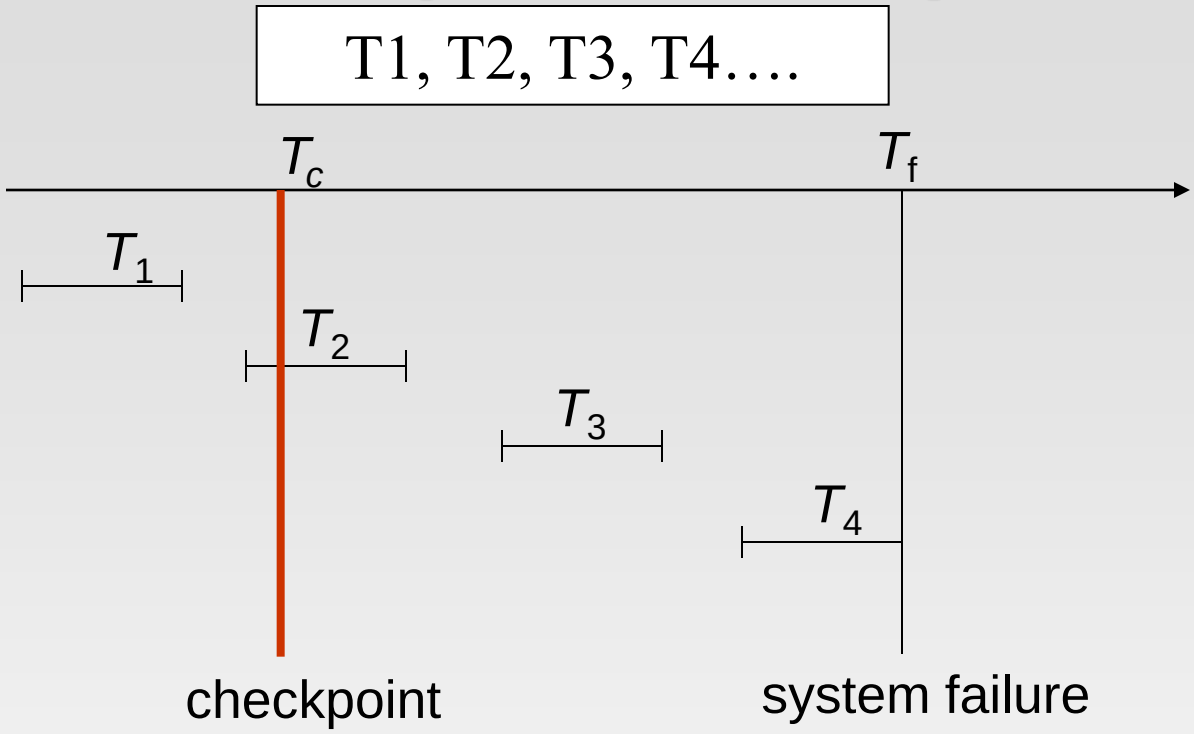


Checkpoints (Cont.)

- During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i.
  1. Scan backwards from end of log to find the most recent <**checkpoint**> record
  2. Continue scanning backwards till a record <T_i **start**> is found.
  3. Need only consider the part of log following above **star**t record.
     1. Earlier part of log can be ignored during recovery, and can be erased whenever desired.
  4. For all transactions (starting from T_i or later) with no <T_i **commit**>, execute **undo(***T_i***)**. (Done only in case of immediate modification.)
  5. Scanning forward in the log, for all transactions starting from T_i or later with a <T_i **commit**>, execute **redo(***T_i***)**.



Example of Checkpoints

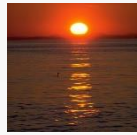


- T_1 can be ignored (updates already output to disk due to checkpoint)
- *Undo* T_4 (remember Undo first, then Redo) // *Redo* T_2 and T_3.

- Be careful for the order of undo & redo
- Suppose T1 (x =3); T2 (x = x + 1); T3 (x = x +1); T4 (x = x * 2)

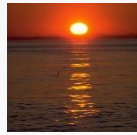
Shadow Paging

- **Shadow paging** is an alternative to log-based recovery
  - this scheme is useful if transactions execute serially
- Idea: maintain *two* page tables during the lifetime of a transaction
  - the **current page table**, and the **shadow page table**
- Store the shadow page table in nonvolatile storage, such that state of the database prior to transaction execution may be recovered.
  - Shadow page table is never modified during execution
- To start with, both the page tables are identical.
  - Only current page table is used for data item accesses during execution of the transaction.
- Whenever any page is about to be written for the first time
  - A copy of this page is made onto an unused page.
  - The current page table is then made to point to the copy
  - The update is performed on the copy

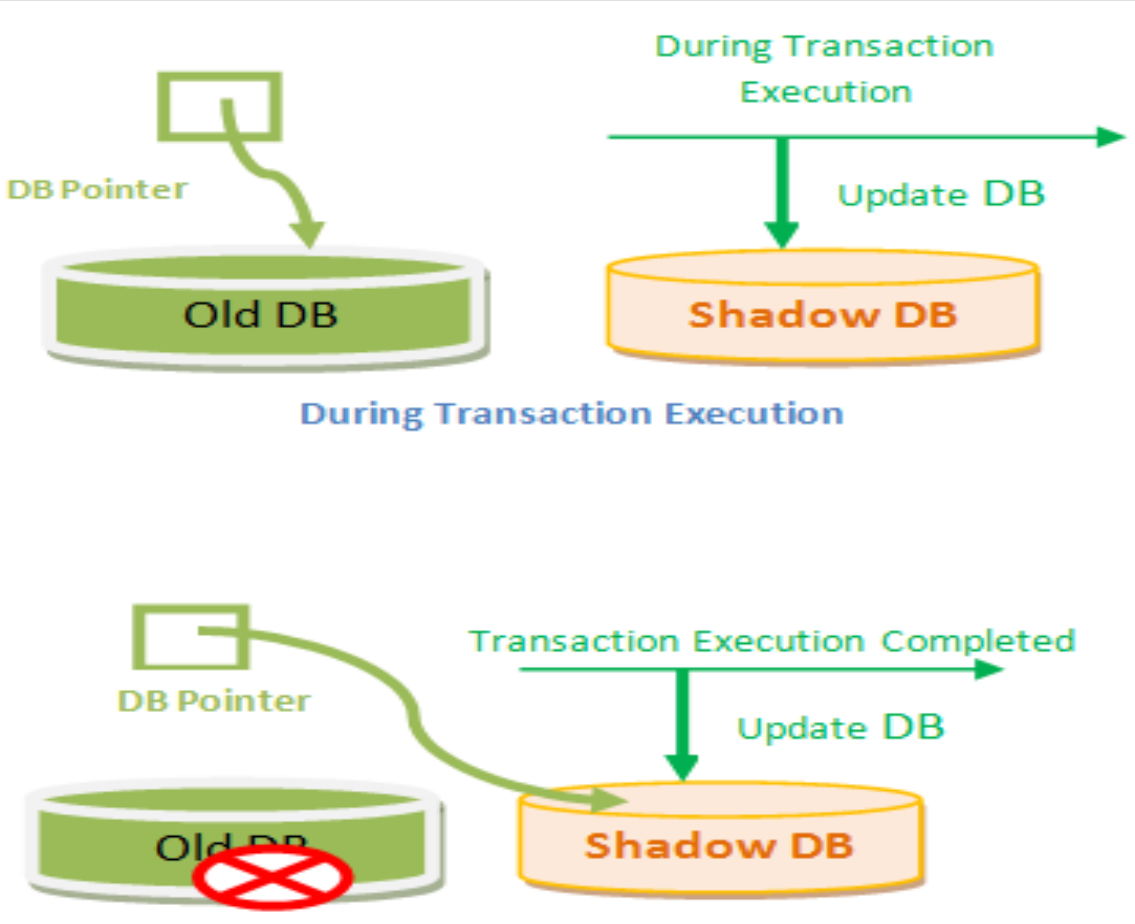


When transaction starts, both page tables are identical

- The shadow page table is never changed over the duration of the transaction.
- The current page table may be changed when a transaction performs a write operation.
- All input and output operations use the current page table to locate database pages on disk.

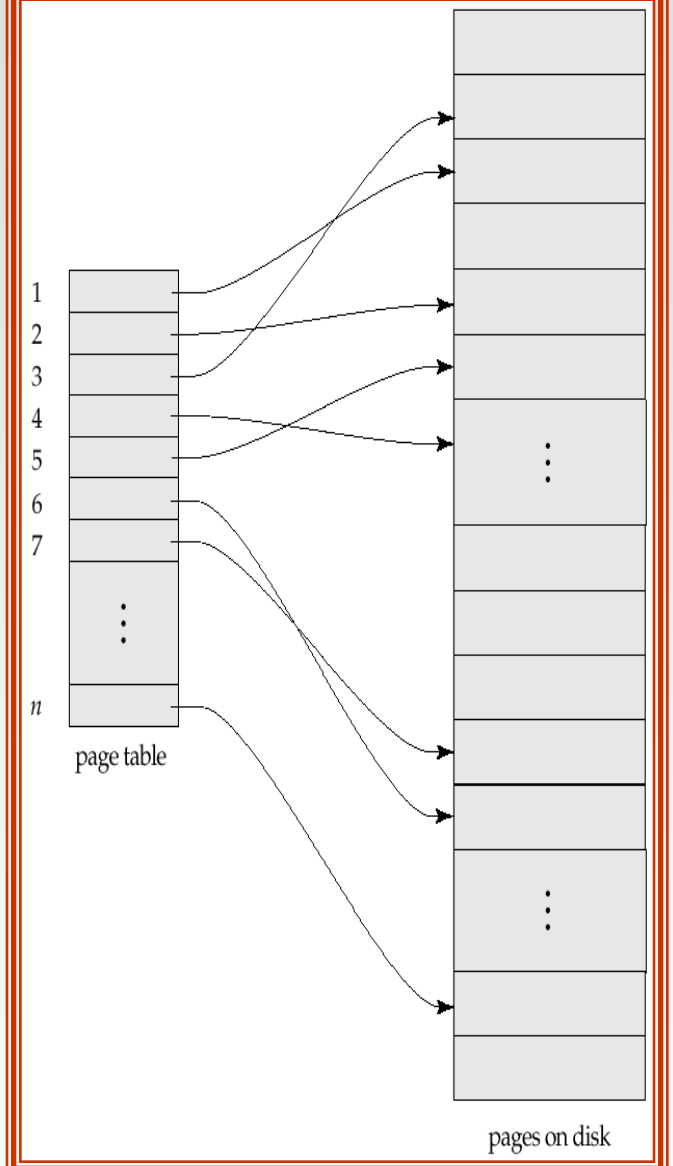


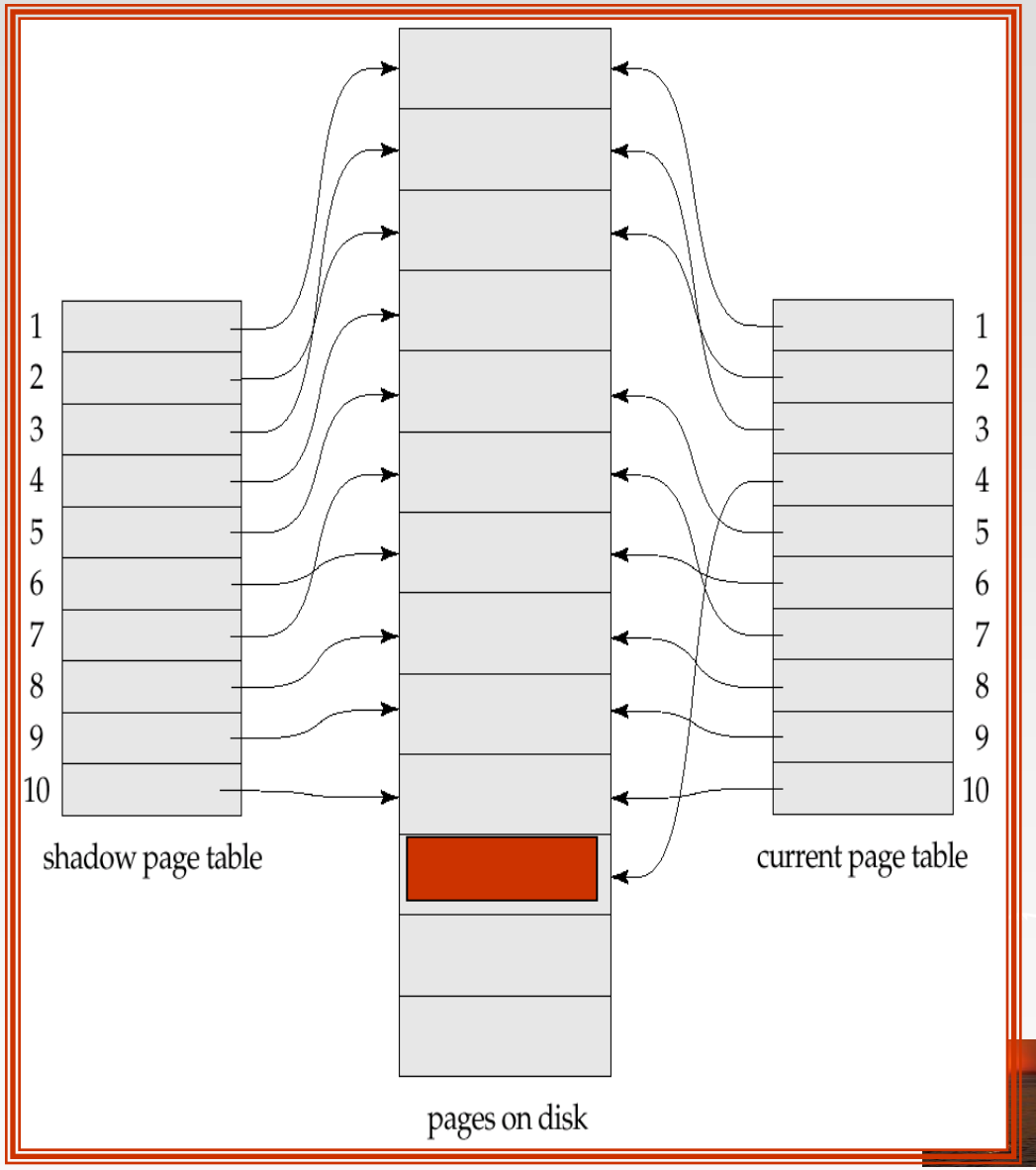
working



Example of Shadow Paging

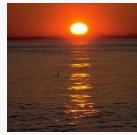
Shadow and current page tables after write to page 4



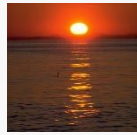


Shadow Paging (Cont.)

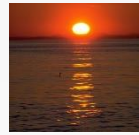
- To commit a transaction :
  1. Flush all modified pages in main memory to disk
  2. Output current page table to disk
  3. Make the current page table the new shadow page table, as follows:
     - keep a pointer to the shadow page table at a fixed (known) location on disk.
     - Simply update the pointer to point to current page table on disk
- Once pointer to shadow page table has been written, transaction is committed.
- No recovery is needed after a crash — new transactions can start right away, using the shadow page table.
- Pages not pointed to from current/shadow page table should be freed (garbage collected).



- A system is in a _____ state if there exists a set of transactions in which every transaction is waiting for another transaction in the set.
  - Deadlock
  - Starved
  - Isolated
  - None of the mentioned
- **What are the ways of dealing with deadlock ?**
  - **Deadlock prevention**
  - **Deadlock recovery**
  - **Deadlock detection**
  - **All of the mentioned**

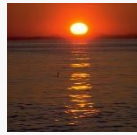


- ……. is an alternative of log based recovery.
  - Disk recovery
  - Shadow paging
  - Dish shadowing
  - Crash recovery
- The _____ protocol releases all locks only at the end of the transactions
  - Graph based protocol.
  - Strict two phase locking protocol.
  - Two phase locking protocol.
  - Rigorous Two phase locking protocol.

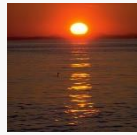


- Validation based protocol is called as _____.
  - pessimistic concurrency control.
  - optimistic concurrency control.
  - right concurrency control.
  - perfect concurrency control

- In which the database can be restored up to the last consistent state after the system failure?
  - Backup
  - Recovery
  - Both
  - None



- In log based recovery, the log is sequence of .....
  - filter
  - records
  - blocks
  - numbers
- If a transaction has obtained a _____ lock, it can both read and write on the item
  - Shared mode
  - Exclusive mode
  - Read only mode
  - Write only mode



- A DBMS uses a transaction ____ to keep track of all transactions that update the database
  - log
  - table
  - block
  - Statement