Recovery System

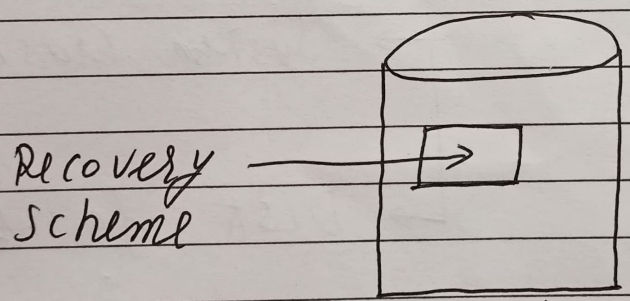
PAGE NO.: 1
12 | 09 | 2022

"Information may be lost" due to "failure of computer" causes disk crash, power outage, software error, fire in the computer room or even sabotage.

→ The database system must take actions in "advance" to ensure the "atomicity" and "durability" properties of transactions.

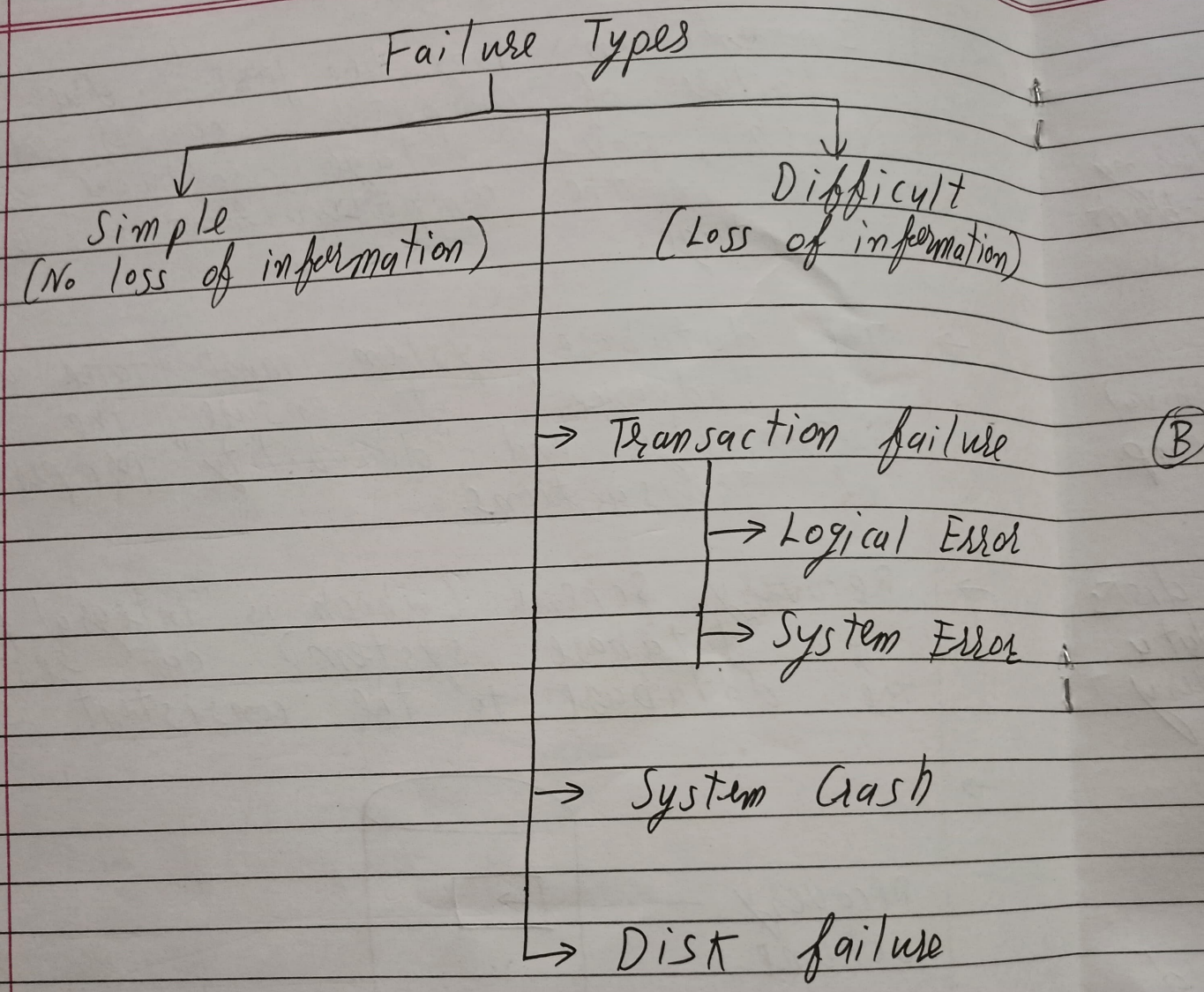
→ Recovery scheme (which is integral part of database system) can restore the database to the consistent state.

→



→ Recovery scheme must restore all ~~important~~ information / data after system crash.

→ Data are important; hence "Recovery System" should provide high availability.



(A) Transaction Failure :

① Logical Error : Due to bad input, data not found, overflow or resource limit, etc.

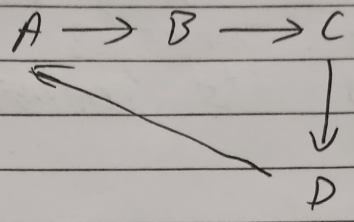
To send 100

Bad input ← But 'ABC', A → B

Minimum =100

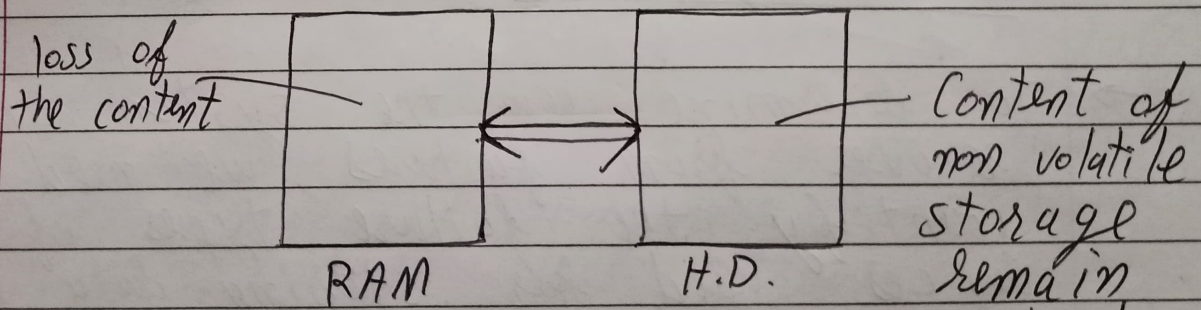
To send 500 → Resource limit.

② System Error : Due to deadlock of transaction. (Re execute in future)



(B) System Crash :

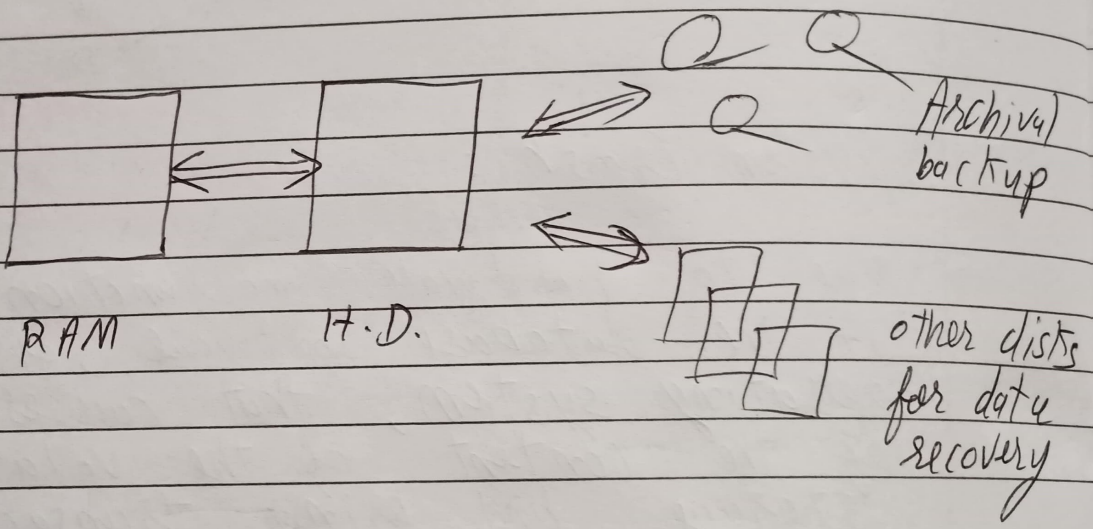
→ Due to hardware malfunction, bug in the database software, or the operating system, that causes the loss of the content of the volatile storage and bring transaction processing to a hault.



→ Fail Stop Assumption : when hardware errors or bugs in the software bring the system to a hault but do not corrupt the non-volatile storage content is known as "Fail Stop Assumption".

(c) Disk Failure :

→ Due to "head crash" or failure during data transfer operation a disk-block loses its content.



Failure Analysis :

→ To determine how the system should recover from failures, we need to identify the failure types of those devices used for storing data and how these failure modes affect the content of the database.

→ Actions are taken based on failure types to maintain the database consistency.

Parts of Recovery Algorithm:

① Before failure: Necessary action should take.

② After failure: Actions should take

Storage Type:

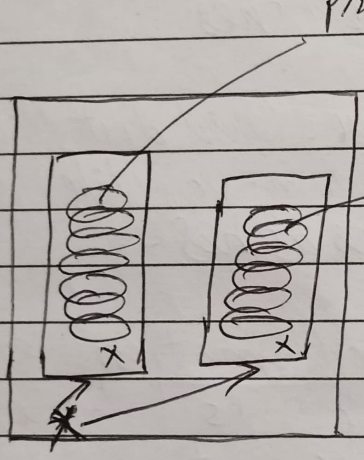
→ Volatile storage: (RAM / cache) Data residing in volatile storage does not usually survive system crashes.

→ Non-Volatile Storage: (Hard disk / Magnetic tapes) Data residing in non-volatile storage survives system crashes.

→ Stable storage: Information residing in stable storage is "never lost". Although pratically it is impossible to obtain, but it can be closely approximated by techniques that make data loss extremly unlikely.

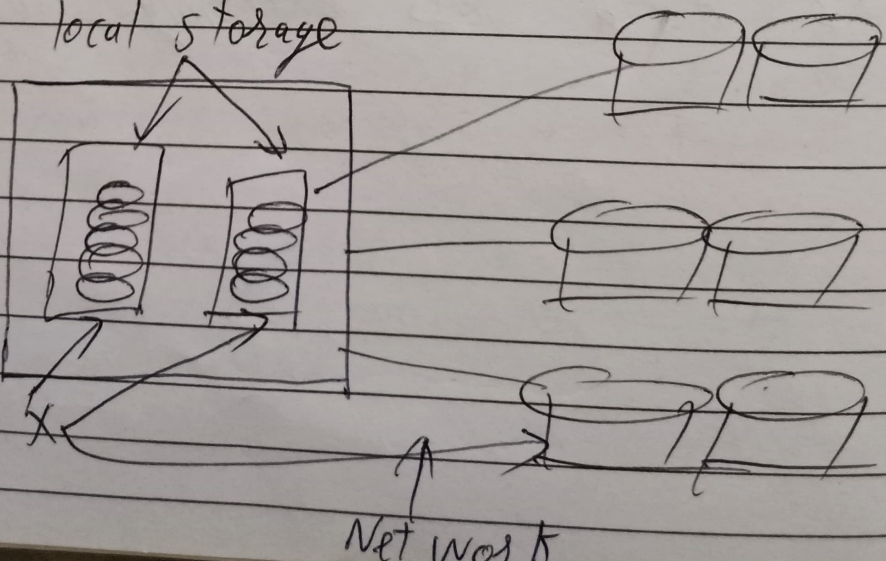
How to implement stable storage?

To implement stable storage, we need to "replicate" the needed information in "several non-volatile storage media" (disk) with independent failure modes, and to update the information in a controlled manner to ensure that failure during data transfer does not damage the needed information.



(cannot guard against disasters such as fire & flooding)

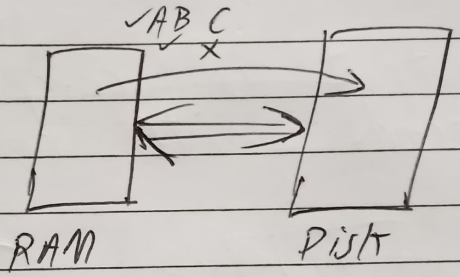
(Redundant arrays of independent disk.)



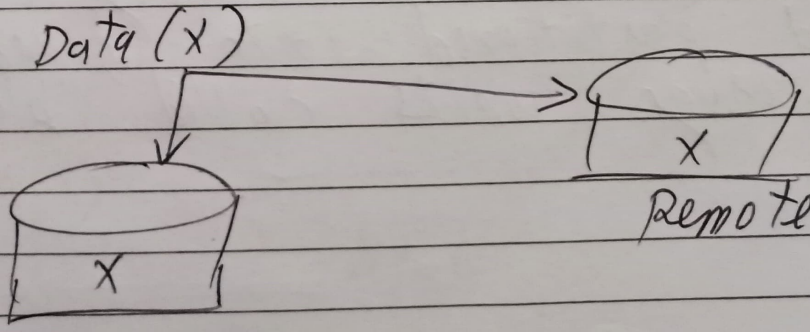
How storage media can be "protected" from "failure" during data transfer?

→ The block transfer between memory & disk storage can result in:-

(a) Successful completion
(b) Partial Failure
(c) Total Failure



→ If a data transfer failure occurs, the system detects it and invokes a recovery procedure to restore the block to a consistent state. To do so the system must maintain two physical block for each logical database block.

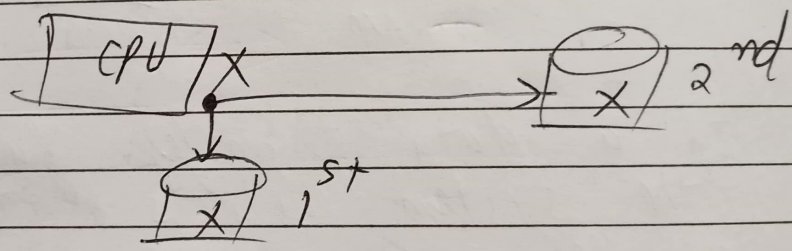


→ An output operation is executed as follows:

① Write the data on to the first physical block.

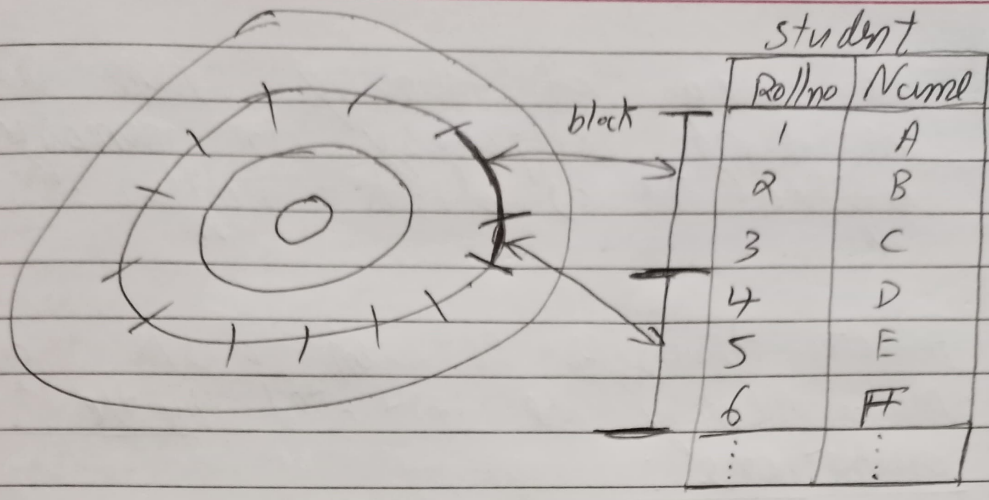
② When the first write completes successfully, write the same data on to the second disk.

③ The output is completed only after the second write completes successfully



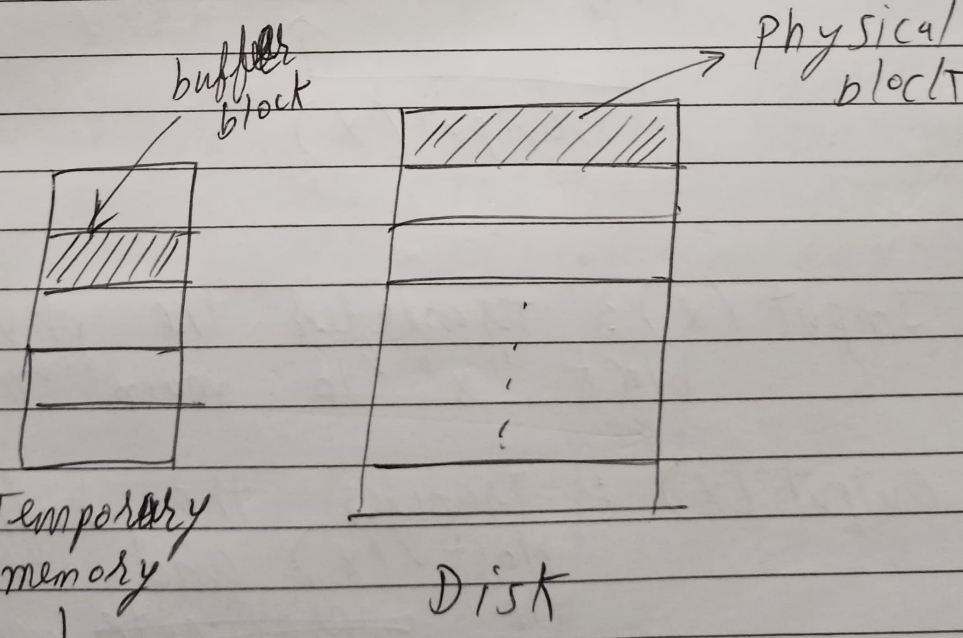
Data Access:

→ The database system resides permanently on non-volatile storage (usually disk), and partitioned into fixed length storage units called "block".



→ Blocks are the units of data transfer to and from disks and may contain several data items.

→ The input & output operations are done in block units. (i.e. block by block, not record by record).

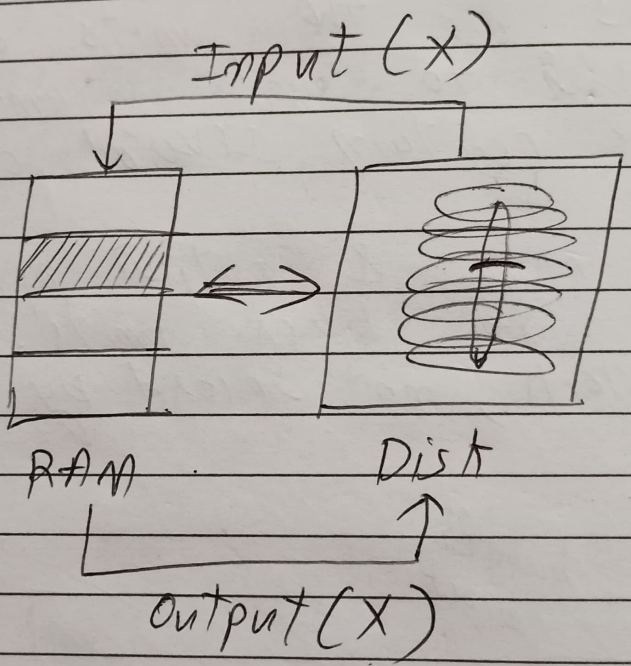


↓

Disk buffer.

→ The area of memory where blocks residing temporary in memory are refered to as disk buffer.

Block transfer between disk and main Memory are initiated by the two operations:



① Input (x): Transfer the physical block "x" to main memory.

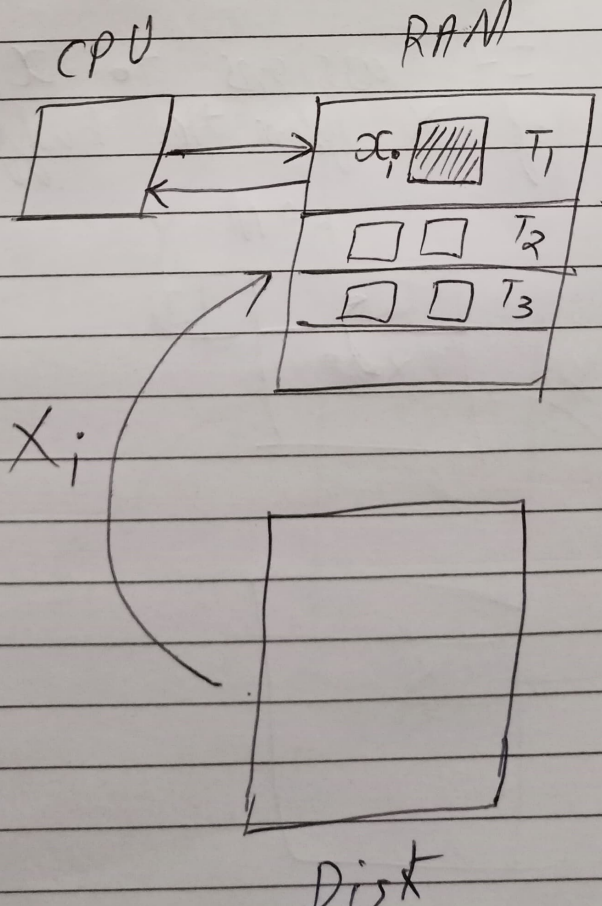
② Output (x): Transfer the buffer block (x) and replace the ~~approximate~~ appropriate physical block there.

Data Management by Transaction :

(1) The system creates a work area (each transaction has a private work area to store all the data accessed / updated by the transactions) when the transaction is initiated.

(2) The system removes it when the transaction either commits or aborts.

(3) Let each data x_i kept in the work area of transaction T_i is denoted by x_i.



④ Transaction T_i interacts with the database system by transferring data to & from its work area.

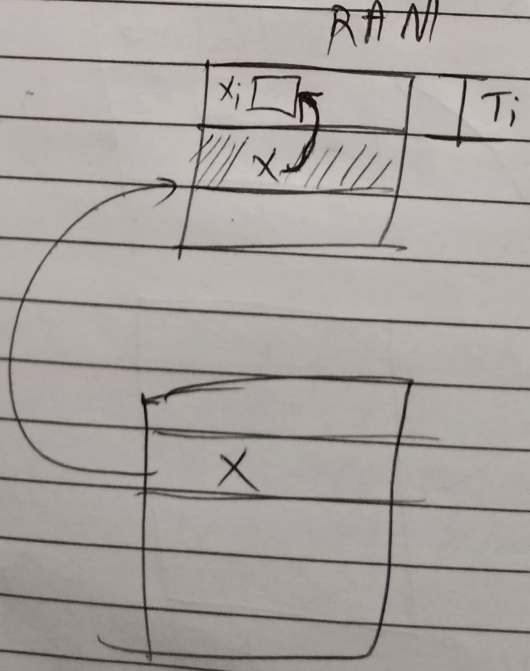
Data Transfer Operations :

① Read (X): Assigning the value of data X to the local variables x_i.

Execution steps:

(a) If block B_x on which X resides is not in main memory, it issues Input (B_x).

(b) It assigns to x_i the value of X from the buffer block.

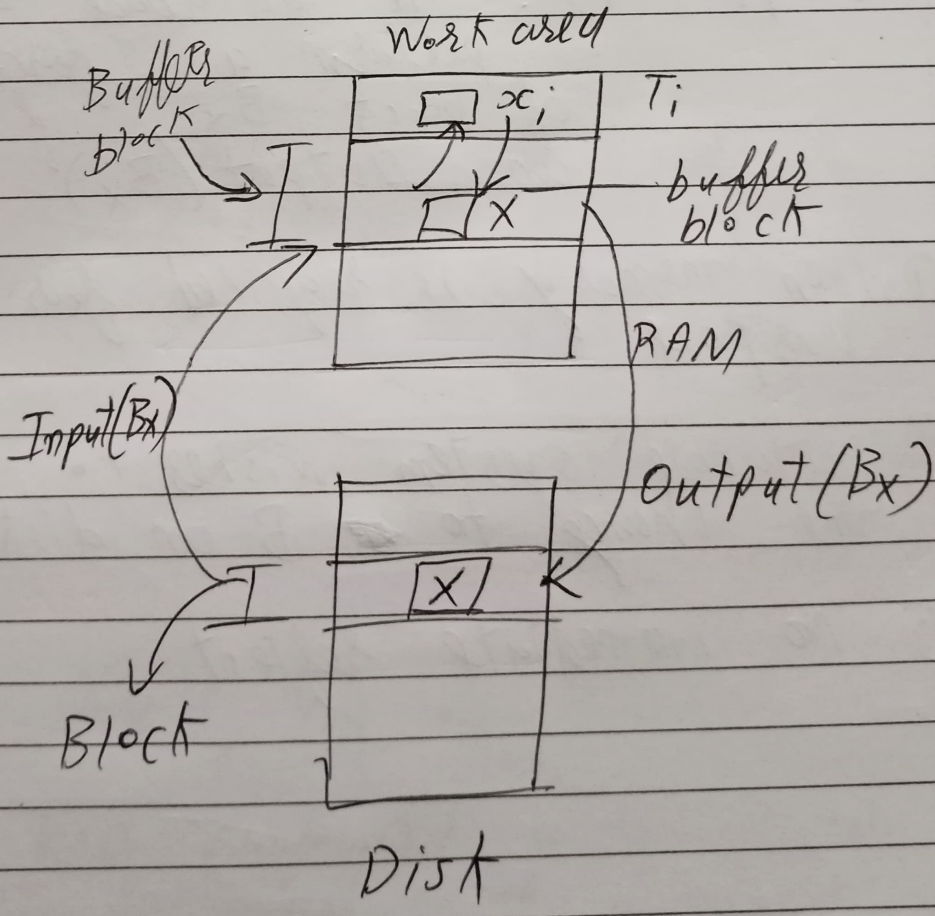


② Write (X): Assigns the value of local variable x_i to data X in the buffer block.

Execution steps:

(a) If block B_X on which X resides is not in main memory, it issues Input (B_X)

(b) It assigns the value of x_i to X in buffer B_X.



→ When a transaction needs to access a data item X for the first time it must execute Read(X).

The system then performs all updates to X on x_i.

After the transaction access X for the final time, it must execute Write(X) to reflect the change to X in the database itself.

③ Output(B_x): The database system perform a "force-output" of block B_x, if it issues an Output(B_x).

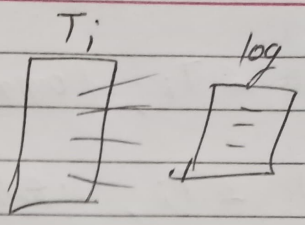
→ ① When memory is required for other purpose

② Database system wishes to reflect the change to B on disk.

③ No immediate reflect.

Log-Based Recovery

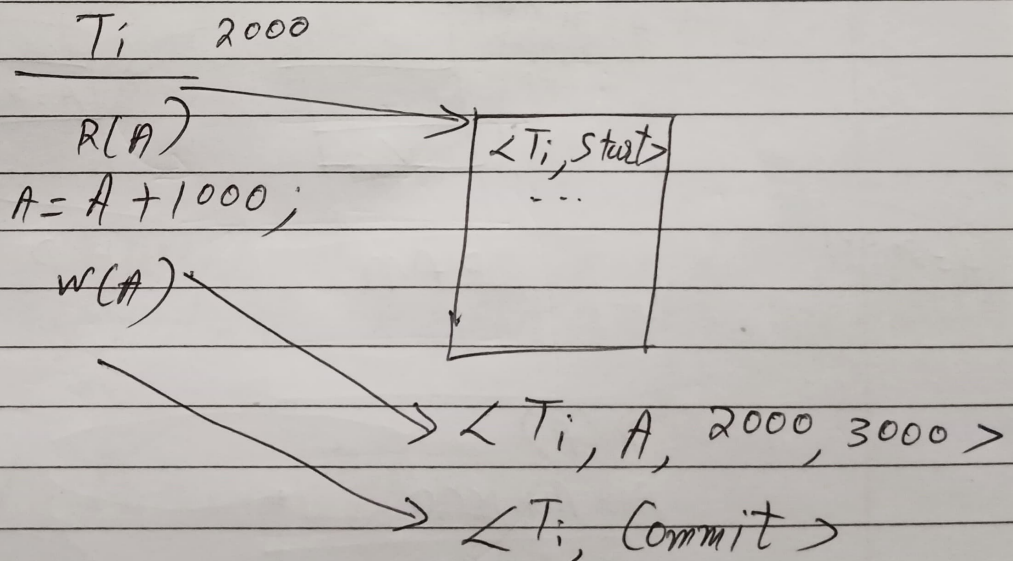
PAGE NO.: 15
14 | 09 | 2022



The log is a sequence of "log records" recording all the update activities in the database. In other words, it is a structure for recording/storing database modifications/activity.

Types of log records are:

(a) $< T_i, start >$: Transaction T_i has started.



(b) $< T_i, data\ j, V_{old}, V_{new} >$:

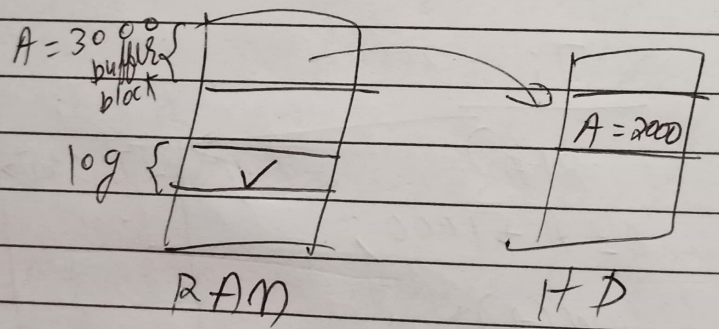
T_i has performed a write on data j, data j had old value "vold" before the write and will have value "Vnew" after the write.

(c) $\langle T_i, \text{commit} \rangle$: T_i has committed.

(d) $\langle T_i, \text{aborted} \rangle$: T_i has aborted.

→ Whenever a transaction perform a "write", it is essential that the log records for that write be created before database is modified.

→ Once a log record is ready we can output the modification to the database if required.



Initial
A = 2000

R(A)
A = A + 1000
W(A)

→ Also, we have the ability to undo a modification that has already been output to the ~~main~~ database. "Undo" can be possible by using the old value field in log record.

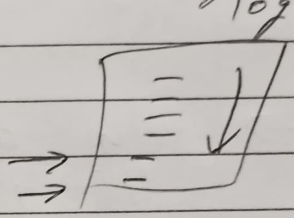
(restoring the old value)

→ Notes on log record.

(a) The log must be stored in stable storage.

(b) It is useful for "recovery" from "system and disk failures".

(c) Every log record is written to the end of the log on "stable storage".



(d) After use it, log record must be deleted from the memory.

(e) Types :-
(i) Deferred database modification
(ii) Immediate database modification

(i) Deferred Database Modification :

→ It ensures transaction atomicity by recording all data base activities in the log, but deferring the execution of all write operation of a transaction until the traction "partially commits". (After final action of the transaction has been executed)

T_i

R(A) //300

A = A + 100; //300

W(A) X → Not reflect here

R(B) //200

B = B - 100; //100

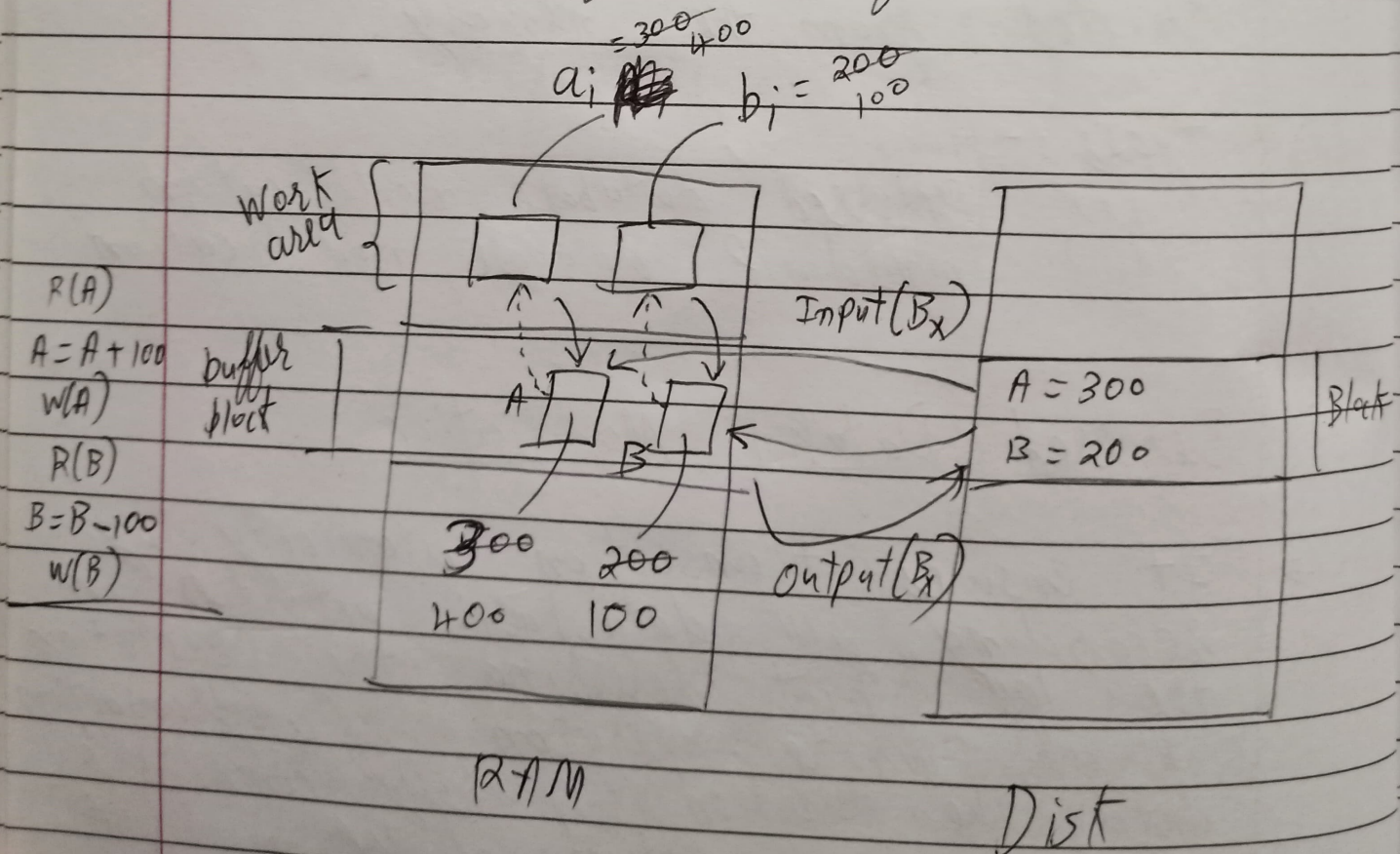
W(B)

Commit → A = 400, B = 100 ✓

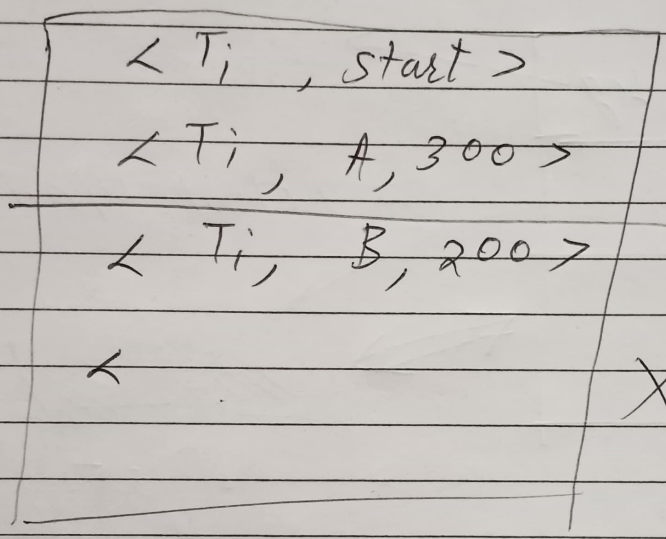
A = ~~300~~

B = ~~200~~

→ When a transaction pastially commits the information on the associated with the transaction is used in executing the deferred writes



→ If the system crashes before the transaction completes its execution, or if the transaction aborts, then the information on the log is simply ignored.



→ Store in Stable storage before output(B)

→ If crashes before R(B)

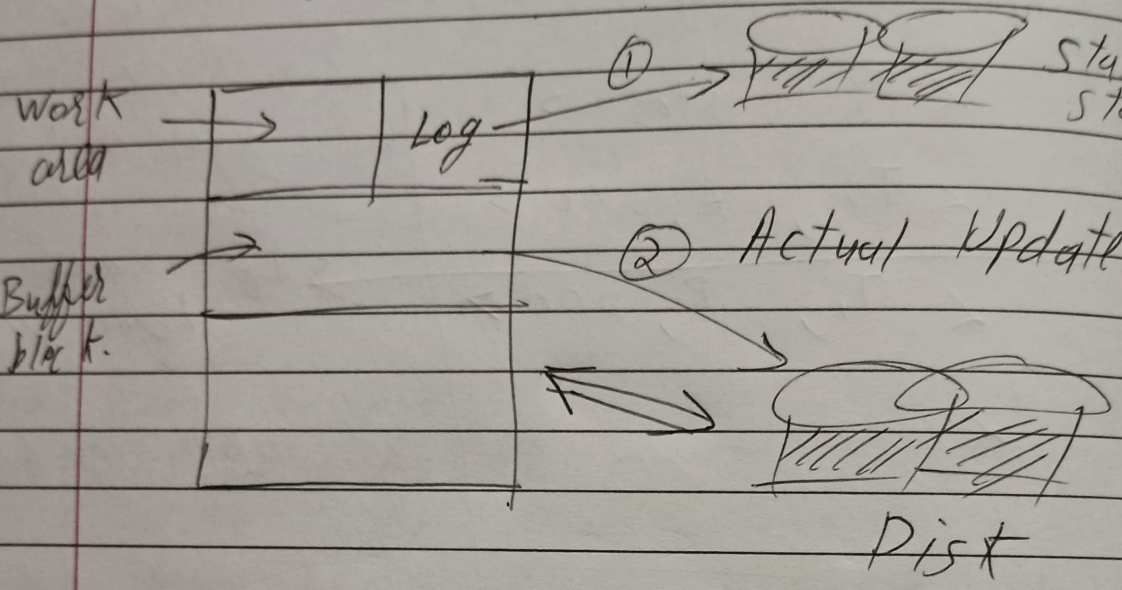
✗ Delete log file

The execution of transaction Ti proceeds as follows.

Ti	Log // Just before start.
	→ < Ti, start >
R(A)	
A = A + 100	
W(A)	→ < Ti, A, 300, 400 >
R(B)	
B = B - 100	→ < Ti, B, 200, 100 >
W(B)	
	→ < Ti, commit >
commit	// Just after partial commit.

Block log

→ when T_i partially commits, the records associated with it in the log are used in executing the deferred writes.



→ Before the start of the update, all the log records are written out to stable storage.

→ Only new value of the data is required by deferred modifications technique.

Example :

A = 500, B = 200, C = 2000.

Log

T_0:
R(A) → <To, start>
A = A - 100
W(A) → <To, A, 400>
R(B)
B = B + 100
W(B) → <To, B, 300>
→ <To, Commit>
Commit (A=400, B=300)

⇒ Suppose these transactions are executed serially.

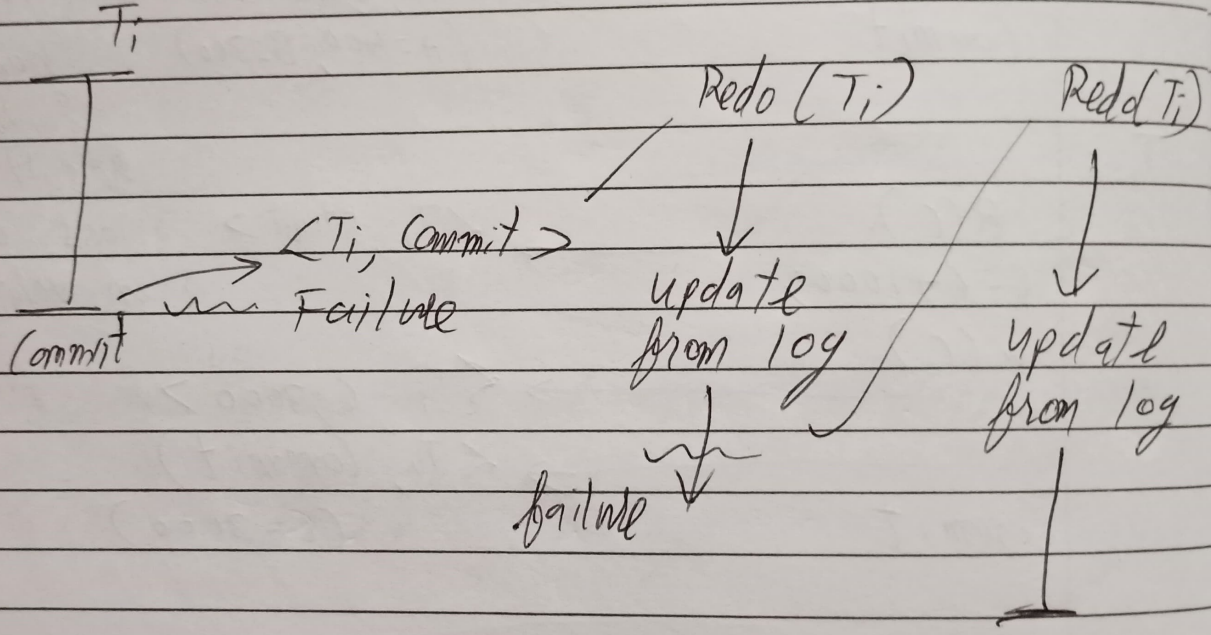
⇒ Using the log the system can handle any failure that result in the loss of data on volatile storage.

T_1:
R(C) → <T1, start>
C = C + 1000;
W(C) → <T1, C, 3000>
→ <T1, Commit)
Commit (C = 3000)

Recovery Procedure :-

Redo (T_i): Sets the value of all data updated by T_i to the new value. For transaction T_i the Redo(T_i) always gives same result. Hence, we can use it as may time as possible.

"Store new value"



→ After a failure, the recovery system consult the log to determine which transaction need to be redone.

→ T_i needs to be redone if and only if log contains both $\langle T_i$, start$\rangle$ and $\langle T_i$, Commit$\rangle$

Example:

A = 500, B = 200, C = 2000

<T0, start>

<T0, A, 400>

<T0, B, 300>

Failure

→ No Redo (T_0) action need to be taken, since no commit record applies in the log.

A = 500, B = 200

<T0, Commit>

<T1, start>

<T1, C, 3000>

Failure

→ Redo (T_0) is performed but no Redo (T_1) is needed.

A = 400, B = 300, C = 2000

<T1, Commit>

Failure

→ Redo (T_0) and Redo (T_1)

A = 400, B = 300, C = 3000

Immediate database Modification:

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

→ Data modification written by active transaction are called "uncomitted modification".

→ In the event of a crash or failure, the system must use the old value field of the log records to restore the modified data to the value they had prior to the start of the transaction.

(a) $\langle T_i, \text{start} \rangle$: Before transaction T_i starts its execution.

(b) $\langle T_i, A, \text{old v}, \text{new v} \rangle$: After a write

(c) $\langle T_i, \text{Commit} \rangle$: When T_i partially committed.

→ Allow the actual update after the log file store into stable storage i.e., just befor Output (B)

< T_0, start >

< T_0, A, 500, 400 >

< T_0, B, 200, 300 >

A = 400, B = 300

< T_0, Commit >

< T_1, start >

< T_1, C, 2000, 3000 >

C = 3000

< T_1, Commit >

→ The recovery scheme uses two recovery procedure:

① Undo (T_i): (Rollback) Restores the value of all the data updated by T_i to the "old value".

② redo (T_i): Sets the value of all data updated by T_i to the ~~value~~ new value

→ After a failure has occurred, the recovery scheme "consults the log" to determine which transaction need to be "redone" and which need to be "undone".

If log contains $\langle T_i, start \rangle$ and $\langle T_i, commit \rangle$

then T_i needs to redo(T_i);

else if log contains $\langle T_i, start \rangle$ but not $\langle T_i, commit \rangle$

then T_i needs to undo(T_i);

Ex:-

$\langle T_0, start \rangle$

$\langle T_0, A, 500, 400 \rangle$ → Undo(T_0)

$\langle T_0, B, 200, 300 \rangle$

Failure

$\langle T_0, commit \rangle$

$\langle T_1, start \rangle$

$\langle T_1, C, 2000, 3000 \rangle$ → Undo(T_1) then redo(T_0)

Failure

$\langle T_1, commit \rangle$

Failure → redo(T_0), redo(T_1)

Check Points:

→ When a system failure occurs, we must check the system log to determine those transaction that need to be redone and those that need to be undone.

→ To check it, need to search the entire system log to determine this information.

→ Two major problems:

① Searching is time consuming.

② Redone takes longer time (but all the updates are completed already)

→ "checkpoint" can be used to reduce such types of overhead.

→ The system periodically perform checkpoints which require the following sequence of action to take place:

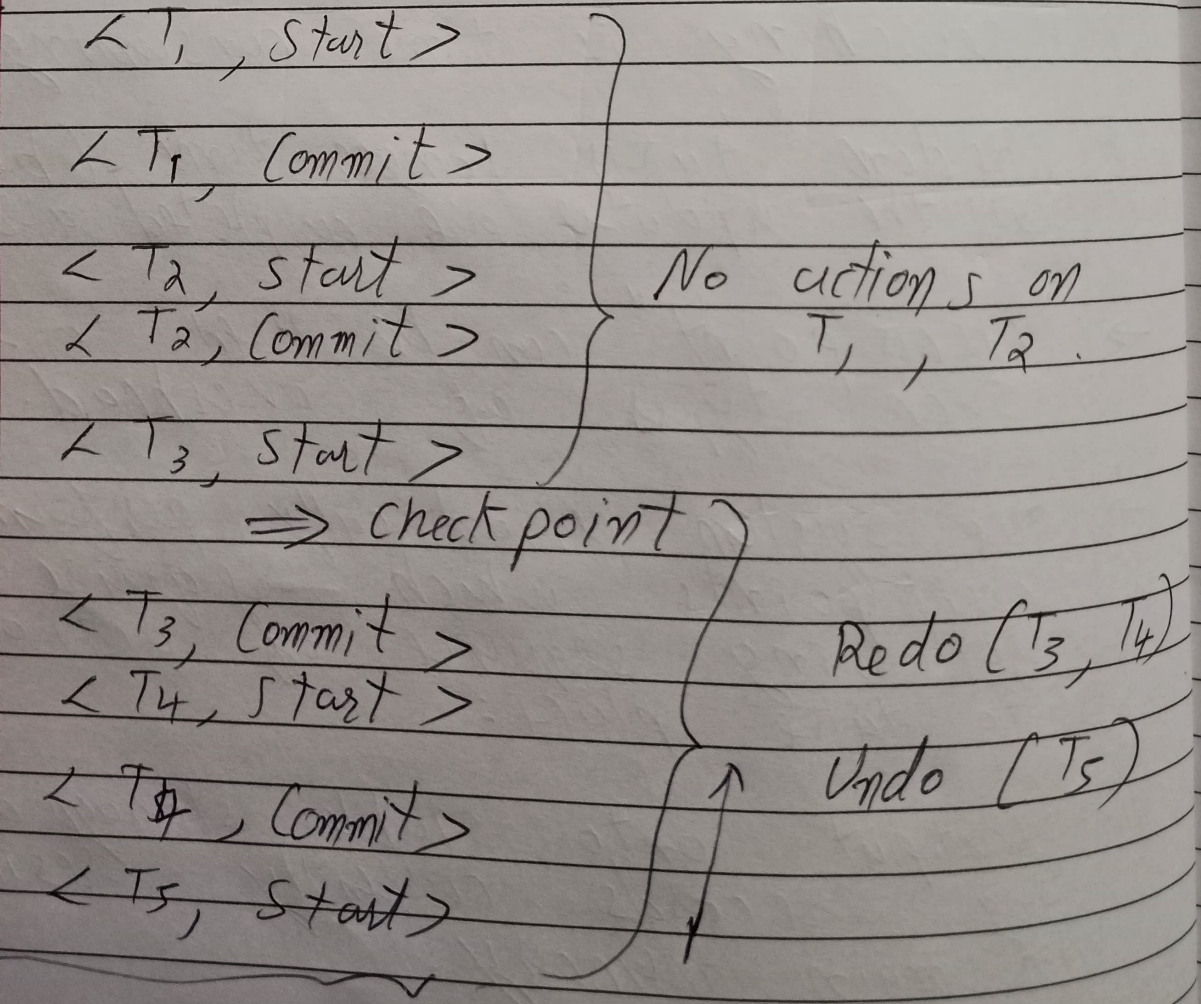
① Output onto stable storage all log records currently residing in main memory.

② Output to the disk all modified buffer blocks.

③ Output onto stable storage a log record <checkpoint>

→ Transactions are not allowed to perform any update actions, such as written to a buffer block or writing a log record, while a checkpoint is in progress.

Ex:



Further

T_0
T_1
T_2
.
.
.
T_{10}
$\Longrightarrow$ checkpoint
T_{11}
T_{12}
T_{13}

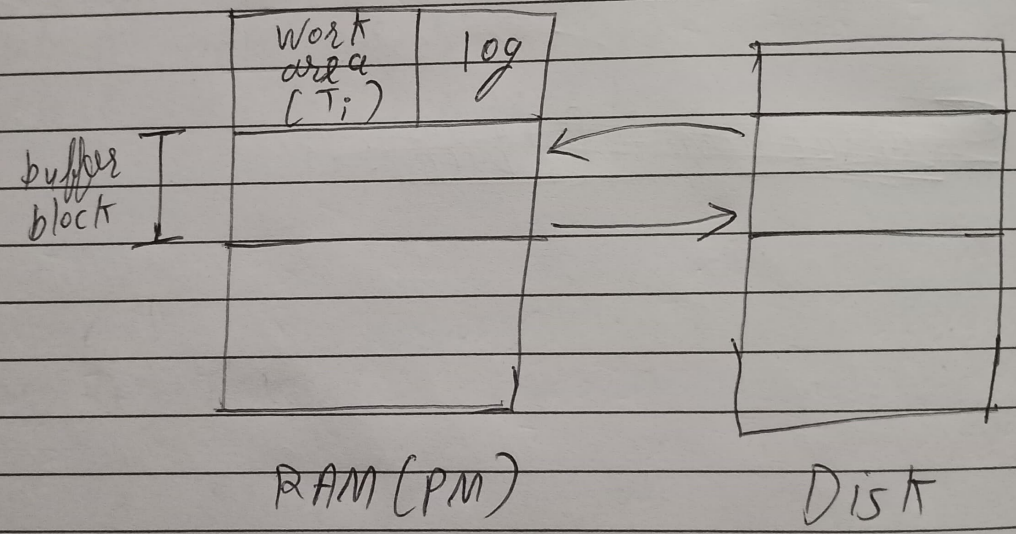
Consider only T_{11}, T_{12}, T_{13} during the recovery system.

~~Transaction Rollback:~~

Concurrent Transaction Recovery

We can use "log" (modified and extended) based recovery scheme to deal with multiple concurrent transaction.

Regardless of the number of concurrent transactions the system has a "single disk" buffer and a "single log".

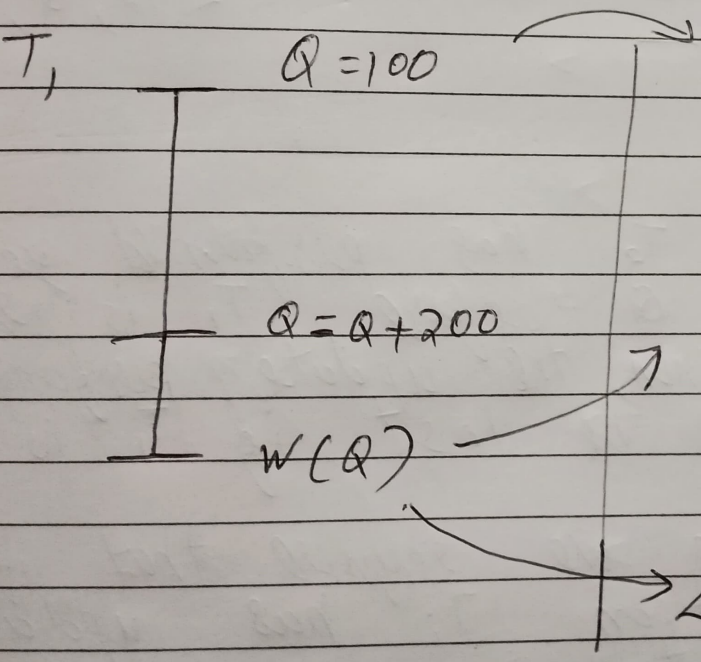


→ All transactions share the buffer blocks. We allow immediate modification and permit a buffer block to have data items updated by one or more transactions.

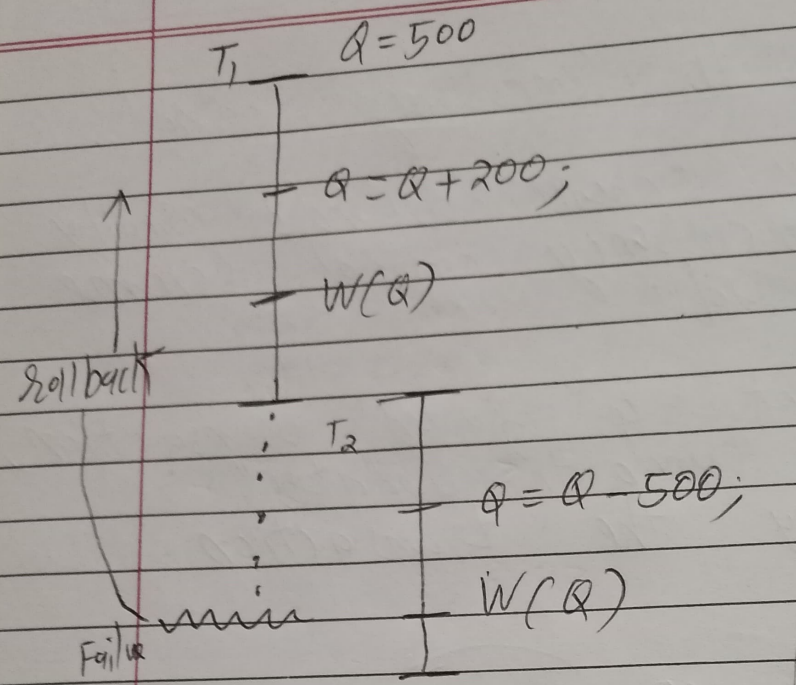
Interaction with Concurrency Control:

The recovery scheme depends greatly on the concurrency control scheme that is used.

→ To roll back a failed transaction, we must "undo the updates" performed by the transaction.



Undo (T_i): To restore the old value



<T1, , Start>

<T1, Q, 500, 700>

<T1, Commit>

<T2, Start>

<T2, Q, 700, 200>

→ Let a T_2 has performed yet another update Q before T_1 is rolled back. Then, the update performed by T_2 will be lost if T_1 is rolled back

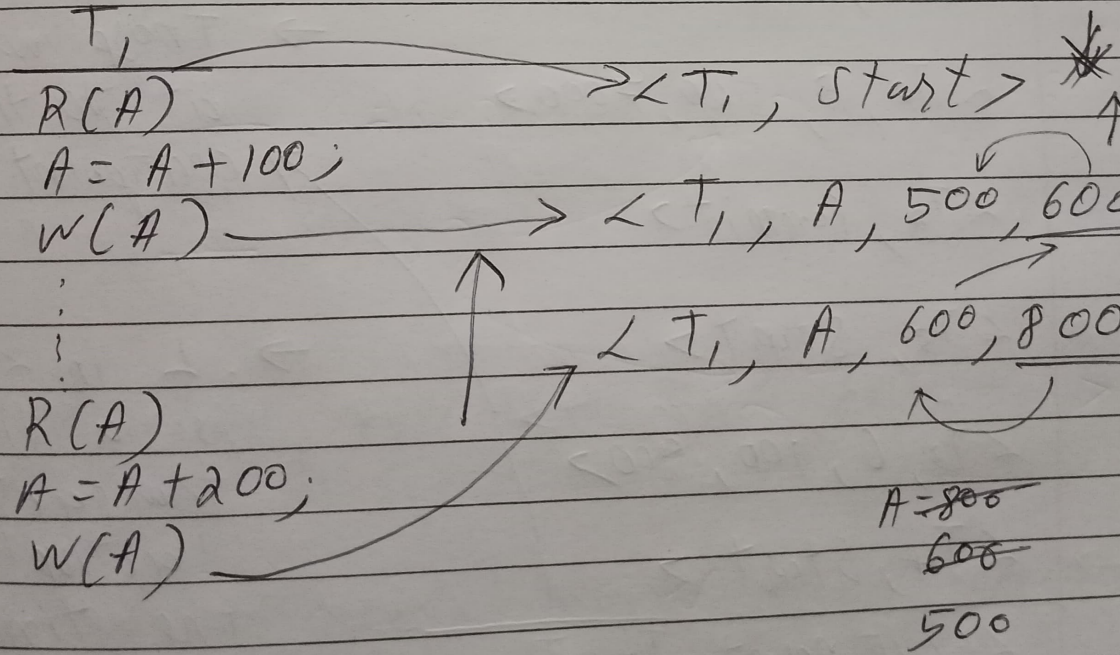
→ Therefore, we require that if a transaction T_i has updated a data Q, no other transaction T_j (where $T_i \neq T_j$) can update the same data until T_i has committed or been rolled back. (We can implement strict two-phase locking).

Transaction Rollback:

→ Using log T_i, can be rolled back. The system scans the log backward; for every log record of the form $\langle T_i, D_j, V_{old}, V_{new} \rangle$ formed in the log.

→ Scanning of the log terminates when the log record $\langle T_i, start \rangle$ is found.

$A = 500$



Checkpoints:

→ We may use check points to reduce the number of log records to scan when it recovers from the crash.

<T1, start>

<T1, A, 100, 200>

<T1, commit>

<T2, start>

<T2, B, 500, 7000>

<T2, commit>

<T3, start>

<T3, C, 100, 900>

<T4, start>

<checkpoint T3, T4>

<T3, commit>

<T4, D, 900, 600>

<T4, commit>

<T5, start>

T_4

T_5

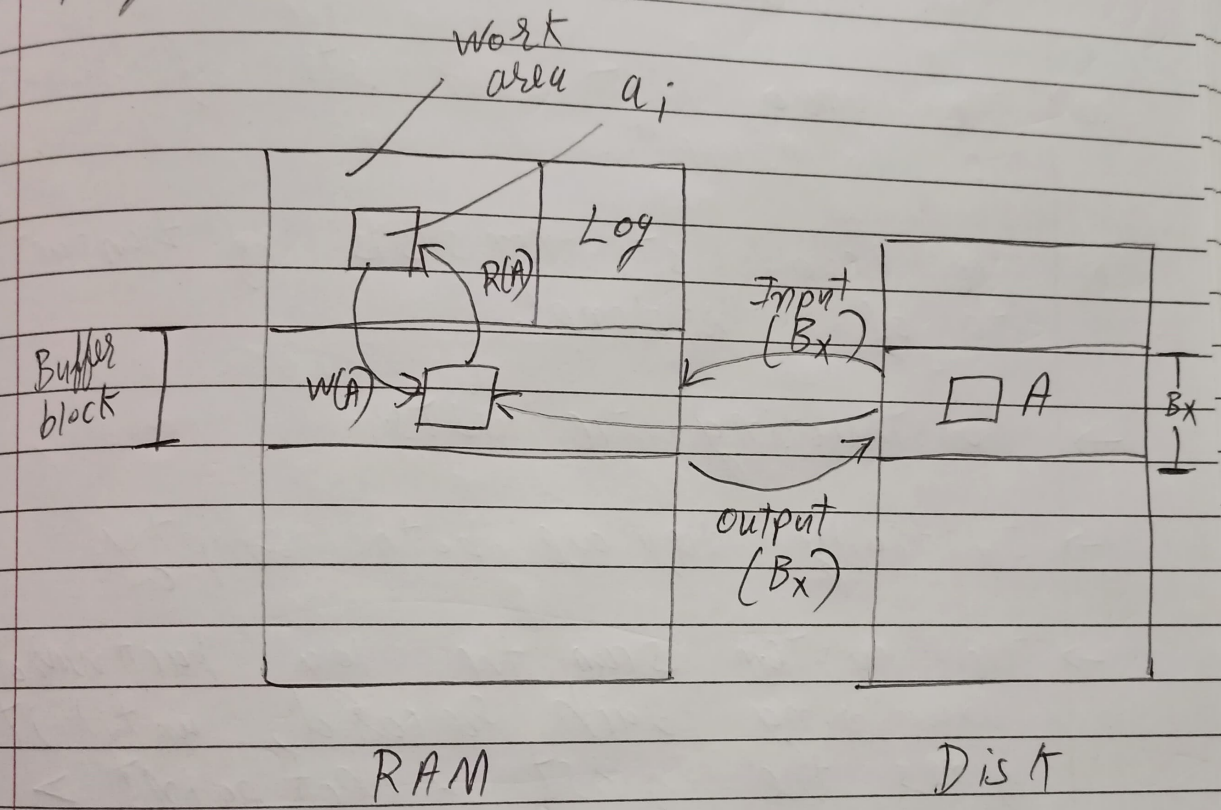
⇒ Consider only those transactions that started after the most recent transaction (T_5)

"OR"

→ Those were active at the time of checkpoint (T_3, T_4)

→ L in <checkpoint L> is a list of transactions "active" at the time of the checkpoint.

→ Transaction "do not" perform update either on the buffer blocks or in the log while check point is in progress.



→ Transaction processing will have to halt while a checkpoint is in progress.

→ A fuzzy check point is a check point where transaction are allowed to perform updates.

Restart Recovery:

To recover from the crash we need two lists:

(a) The undo list consists of transactions to "undone".

(b) The redo list consists of transactions to be "redone".

→ Constructing the lists:

→ Initially, they are both empty.

→ The system scan the log backward examining each record, until it finds the first <checkpoint>

→ For each record found of the form "<Ti, Commit>", it adds 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.

→ When the system has examined all the log records (upto checkpoint), it checks the list L in the checkpoint.

Log :

<T11, Start>
<T12, Start>
<T1, start>
<T1, Commit>
<check point L>

<T2, start>
<T2, Commit>
<T3, start>
<T4, Start>
<T4, Commit>
<T5, start>
<T6, Start>
<T6, Commit>
<T11, Commit>

L = T11, T12

redo list: T11, T6, T4, T2

Undo List: T5, T3, T12

For each T_i in L, if T_i is not in redo List, then it adds T_i to the Undo List.

Continued

After constructing the redo and undo lists, the recovery proceeds as follows:

(a) The system rescan the log from the most recent backward and perform an undo for each log record that belongs to T_i as the undo List.

At that time all redo are ignored. For every record, scan stops when the $<T_i, Start>$ records have been found for every T_i in the undo list.

① Undo $\{T_5, T_3, T_{12}\}$

backward

→ Roll back

② redo $\{T_{11}, T_6, T_4, T_2\}$

forward ← ④ ③ ② ①

(b) Next locate the most recent <checkpoint L> record on the Log. Scan the log forward.

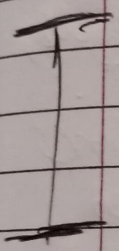
(c) Scan the log forward from the most recent <checkpoint L> record and perform redo all T_i, $T_i \in$ <Redolist>
Now ignore the {Undo List}

(d) First apply Undo and then redo.

Ex:

A = 100 → Initial

Redo{T_2}, Undo{T_1}



$\langle T_1, A, 100, 200 \rangle$

$\langle T_2, A, 100, 300 \rangle$

$\langle T_2, Commit \rangle$

successfull

A = 300

↓

Final

Redo(T_2)	Undo(T_1)
Undo(T_1)	Redo(T_2)

A = 100 → 300 → A

A ⇒ 200 → 100

T_1: A ⇒ 200 → 100

T_2: A ⇒ 100 → 300

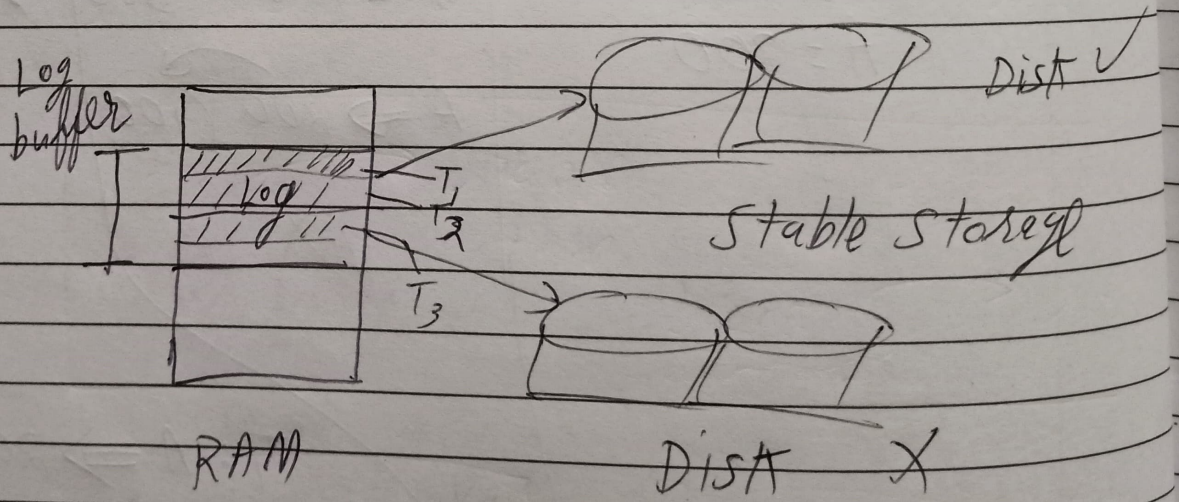
Buffer Management

To implement crash-recovery system, that ensure data consistency, we should have essential concept of buffer Management.

(a) Log record buffering
(b) Database buffering
(c) Role of OS in buffer management.

(a) Log record buffering:

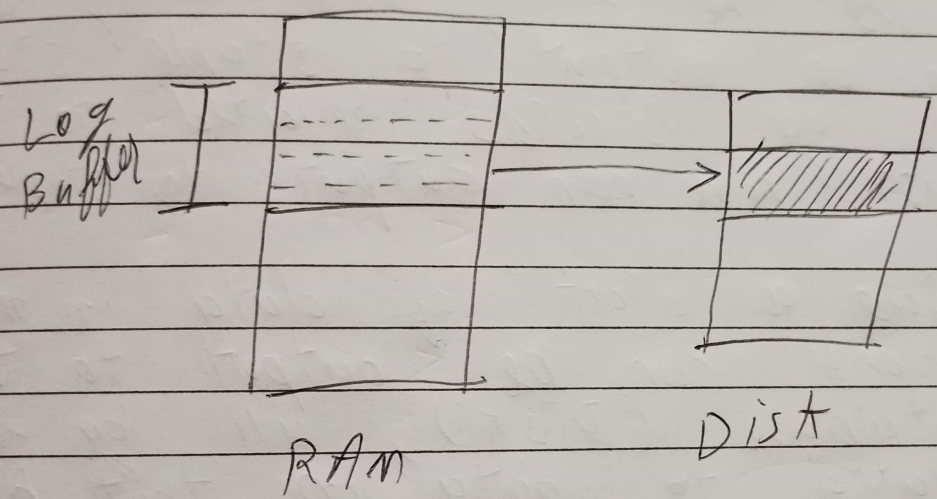
At the time of creation of log buffer, it must store in stable storage. But it is wastage of space, since output of stable storage is in block wise and in most cases, a log record is much smaller than a block.



→ The cost of outputting a block to stable storage is sufficiently high that it is desirable to output multiple log records at once.

→ So, we write log records to a log buffer in main memory temporarily and then store it to stable storage.

→ Multiple log records can be gathered in the log buffer, and output to stable storage in a single output operation.



→ In log buffer, log records may reside in only main memory for a considerable duration before it is output to stable storage. Since such log records are lost if the system crashes we must impose additional requirements on the recovery techniques to ensure transaction atomicity:
(It is called write-ahead logging (WAL) rule)

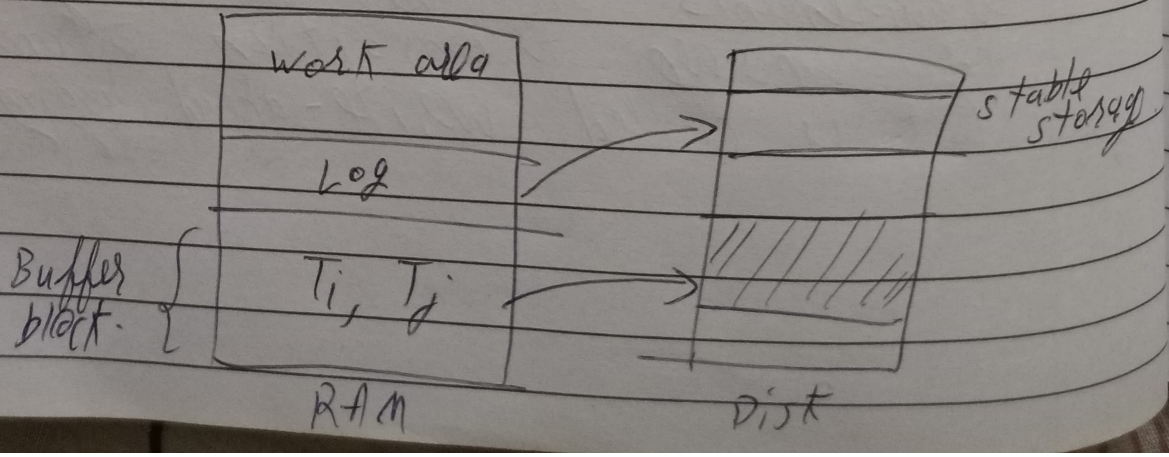
① T_i enters the commit state after the $\langle T_i, Commit \rangle$ log records has been output to stable storage.

Log → stable then
Commit T_i

② Before the $\langle T_i, Commit \rangle$ log records can be output to stable storage, all log records related to T_i must have been output to stable storage.

③ Before a block of data in main memory can be output to the database (Disk), all log records related to data in that block must have been output to stable storage

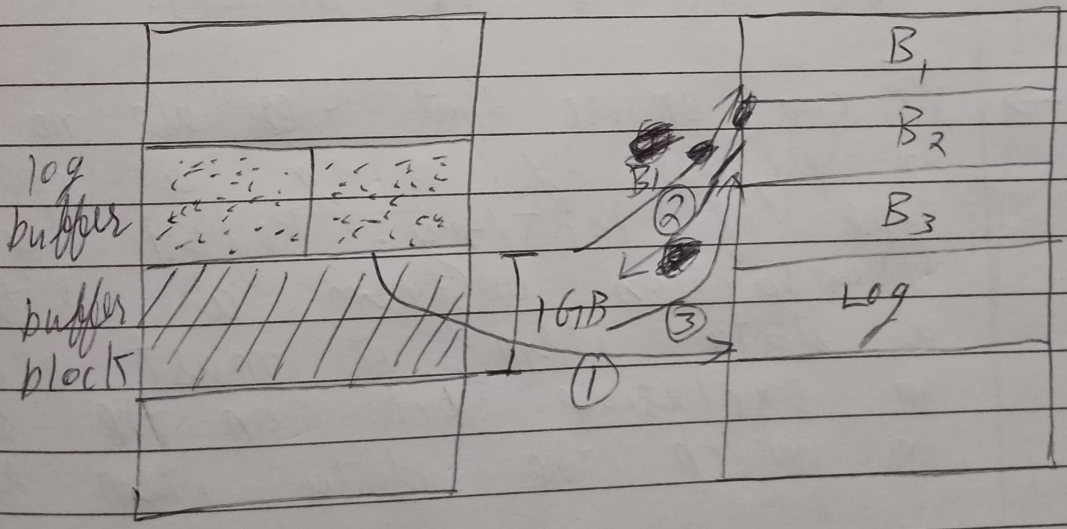
Log → stable
then
block → Disk



→ When the system finds it necessary to output a log record to stable storage, it output an entire block of log records, if there are enough log records in main memory to fill a block.

→ If there are insufficient log records to fill the block, all log records in main memory are combined into a partially full block, and are output to stable storage.

Database Buffering:



→ If the input of block B_2 causes block B_1 to be chosen for output, all log records related to B_1 must be store into stable storage before B_1 is output.

→ The sequence should be as follows:

(a) Output log records to stable storage until all log records related to B_1 have been output.

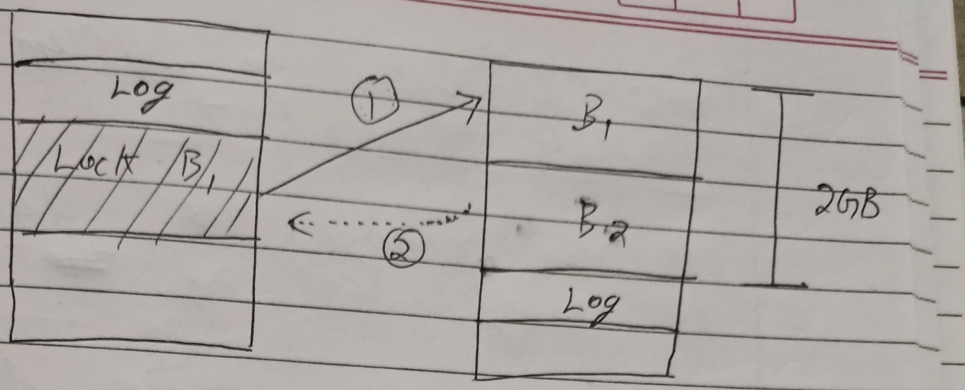
(b) Output block B_1 to disk.

(c) Input block B_2 from disk to main memory.

→ No update to B_1 be in progress while the system carries out this sequence of actions.

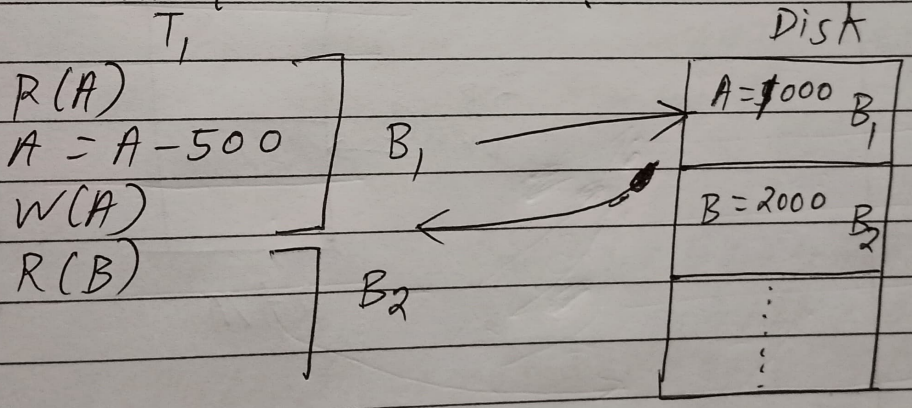
→ We can ensure that there are no updates in progress by using a special type of locking: Before a transaction perform a write on a data item, it must acquire an exclusive lock on the block in which the data item resides.

→ The lock can be released immediately after the update has been performed.



→ Before a block is output, the system obtain an exclusive lock on the block to ensure that no transaction is updating the block.

→ It releases the lock once the block output has completed. It is called "Latch".



→ No space available to store B_2 block where B = 2000 is there, so transfer B_1 from RAM to disk, but if system crash the database will be in inconsistent state. So, before that store $< T_1, A, 1000, 500 >$ in stable storage.

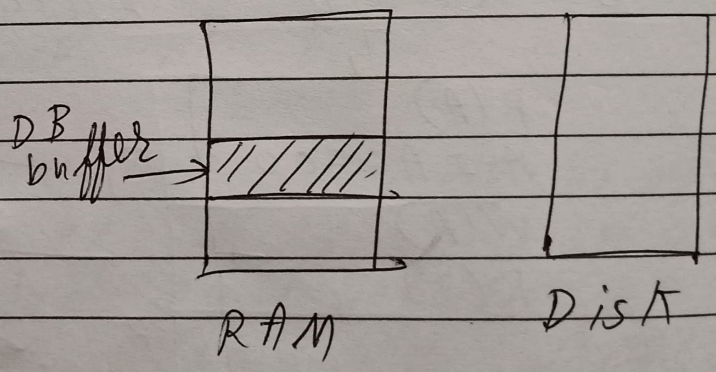
Role of Operating System in Buffer Management :

We can manage the database buffer by using one of two approaches.

① The database system reserve part of Main Memory to serve as a buffer that it rather than the OS Manages.

DBase RAM

The dabase system manages data block transfer in accordance with the requirements.



→ This approach has the drawback of limiting flexibility in the use of main memory.

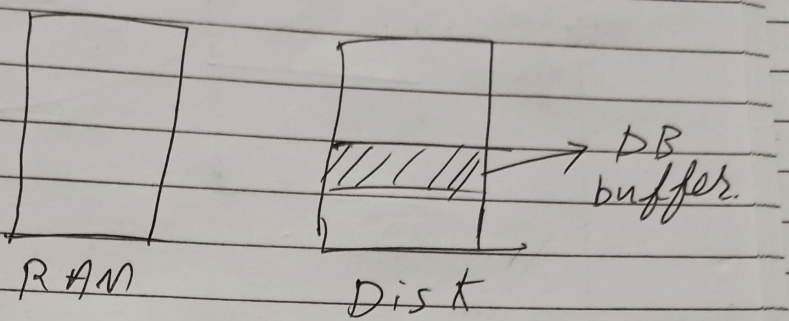
→ The buffer must be kept small enough that other applications have sufficient main memory available of their needs.

OS

Disk

②

② The database system implements its buffer within the virtual memory provided by the OS.

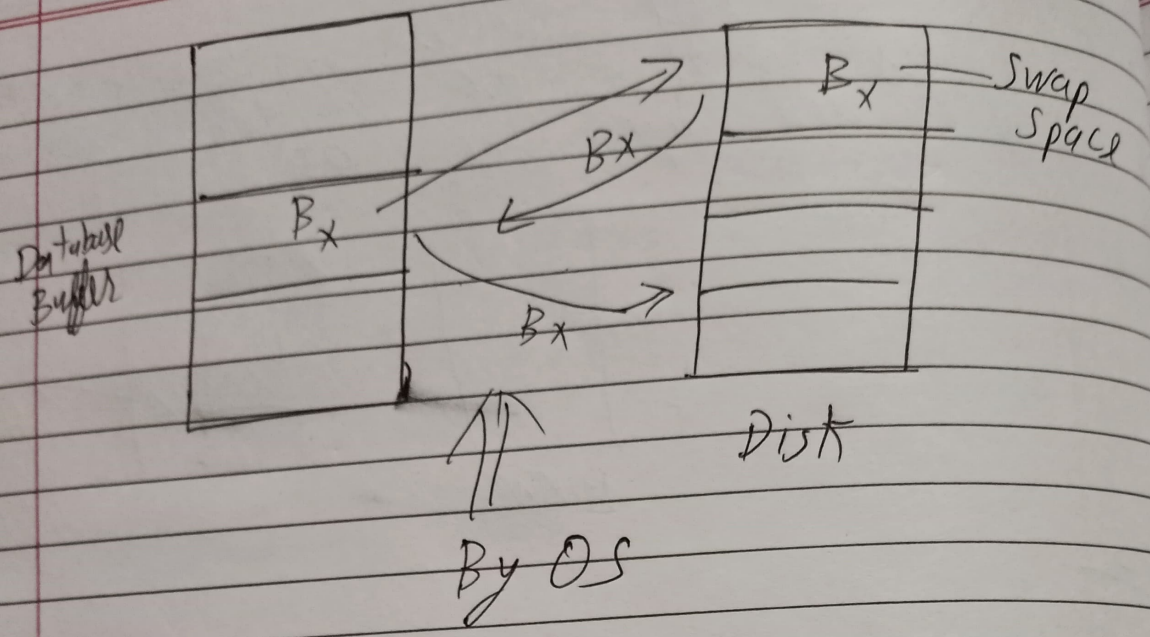


OS

Disk The OS knowns about the memory requirement of all processes in the system, ideally it should be in charge of deciding what buffer block must be force-output to dist and when.

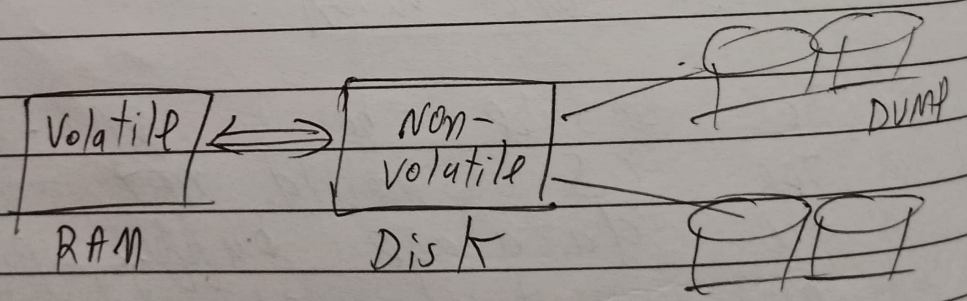
→ To ensure the "WAL-requirement" the OS should not write out the database buffer pages itself, but instead should request the database system to force output the buffer blocks.

→ The database system in turn would force-Output the buffer blocks to the database after writing relevant log records to stable storage



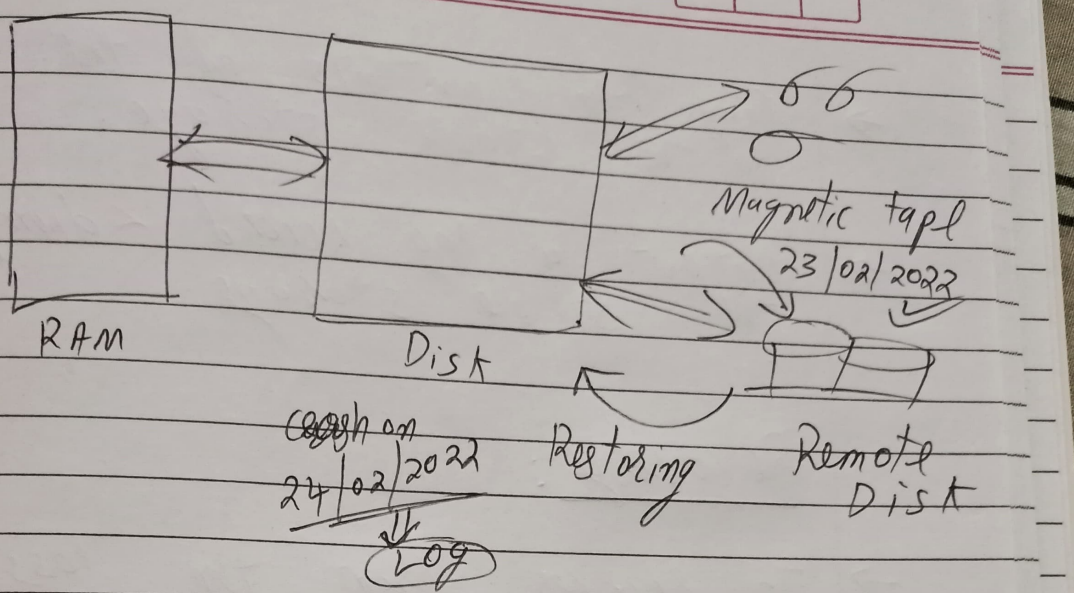
Failure with loss of non-volatile storage:

Failure in which the content of non-volatile storage is lost are rare.



Dump the entire contents of the database to stable storage periodically - e.g. once / day.

We may dump the database to one or more magnetic tapes.



→ If a failure occurs that results in the loss of physical database blocks, the system use the most recent dump in restoring the database to a previous consistent state.

→ After that, the system uses the log to bring the database system to the most recent consistent state.

Dump procedure:

No transaction may be active during the "dump procedure".

A following procedure is adopted:

→ Output all log records currently residing in main memory onto stable storage.

→ Output all buffer blocks on to the disk.

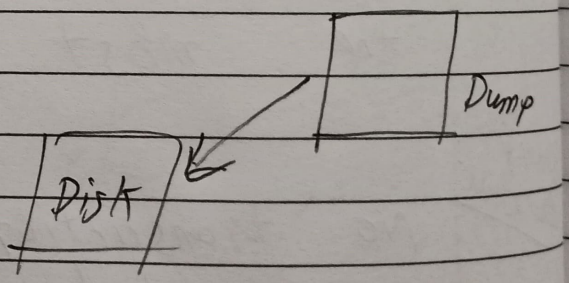
→ Copy the contents of the database to the stable storage.

→ Output a log record <dump> on to the stable storage.

Recovery:

→ To recover from the loss of "non-volatile storage" the system "restores" the database to "disk" by using the most recent dump.

→ It consults the log and redoes all the transactions that have comitted. No undo operation need to be executed.



→ The simple dump procedure described above is "costly" for the following two reasons:

(a) Huge data transfer: The entire database must be copied to stable storage.

(b) CPU cycle are wasted : Since transaction processing is halted during the dump procedure.

"Fuzzy dump scheme" have been developed which allow transactions to be active while the dump is in progress.