Chapter 8:
Crash Recovery

8.1 Failure Classification
8.2 Recovery and Atomicity
8.3 Log-based Recovery
8.4 Shadow paging
8.5 Advanced Recovery Techniques.

8.1 Failure Classification:

To find that where the problem has occured, we generalized a failure into following categories.

i. Transaction Failure.
ii. System crash.
iii. Disk failure.

Transaction failure:

The transaction failure occurs when it fails to execute or when it reaches a point from where it cannot go any further. If a few transaction or process is hurt, then this is called as transaction failure.

Reasons for transaction failure could be-

i. Logical errors:

If a transaction cannot complete due to some code error or an internal error or an internal error condition, then the logical error occurs.

ii. Syntax error:

It occurs where the DBMS itself terminates an active transaction because the database system is not able to execute it. For example: The system aborts an active transaction, incase of deadlock or resource unavailability.

System Crash:

System crash can occur due to power failure or other hardware or software failure.

Example: Operating system error.

Fail stop assumption:

↳ In the system crash, non volatile storage is assumed not to be corrupted.

↳ Database systems have numerous integrity checks to prevent corruption of disk data.

Disk failure:

↳ It occurs where hard-disk drives or storage drives used to fail frequently. It was a common problem in the early days of technology evaluation.

↳ Disk failure occurs due to the formation of bad sectors, disk head crash, and unreachability to the disk

or any other failure, which destroy all or part of the disk storage.

Recovery and Atomicity:

↳ Recovery algorithms are techniques to ensure database consistency and transaction atomicity & durability despite failures.

↳ Recovery algorithms have two parts:

i. Actions taken during normal transaction processing to ensure enough information exists to recover from failures.

ii. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability.

↳ When a system crashes, it may have several transactions being executed and various files opened for them to modify the data items. Transactions are made of various operations, which are atomic in nature. But according to ACID properties of DBMS, atomicity of transactions as a whole must be maintained, that is, either all the operations are executed or none.

↳ When a DBMS recovers from a crash

it should maintain the following:

i. It should check the states of all the transactions, which were being executed.

ii. A transaction may be in the middle of some operation, the DBMS must ensure the atomicity of the transactions in this case

iii. It should check whether the transactions can be completed now or it needs to be rolled back.

iv. No transactions would be allowed to leave the DBMS in an inconsistent state.

Log-Based Recovery:

↳ The log is a sequence of records. Log of each transaction is maintained in some stable storage so that if any failure occurs, then it can be recovered from there.

↳ If any operation is performed on the database, then it will be recorded in the log.

↳ But the process of storing the logs should be done before the actual transaction

is applied in the database.

↳ When transaction T_i starts, it registers itself by writing a $\langle T_i \text{ start} \rangle$ log record.

↳ Before T_i executes write(X), a log record $\langle T_i, X, V_1, V_2 \rangle$ 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_j. X_j had value V_1 before the write, and will have value V_2 after the write.

↳ When T_i finishes its last statement, the log record $\langle T_i \text{ commit} \rangle$ is written.

↳ We assume that log records are written directly to stable storage (that is, they are not buffered)

Let us assume that, there is a transaction to modify the city of a student. The following logs are written for this transaction.

When the transaction is initiated, then it writes 'start' log.

$\langle T_n, \text{start} \rangle$

When the transaction modifies the city from 'Pokhara', then another log is written to the file.

< Tn, city, 'ktm', 'pokhara' >

When the transactim is finished, then it writes another log to indicate the end of the transaction.

< Tn, commit >

There are two approach to use logs:

i. Deferred database modification.
ii. Immediate database modification.

i. Deferred database modification.

↳ The deferred modification technique occurs if the transaction does not modify the database until it has committed.

↳ In this method, all the logs are created & stored in the stable storage, and the database is updated when a transaction commits.

↳ The deferred database modification scheme records all modifications to the log, but defers all the writes to after partial commit.

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

↳ Deferred database modification technique ensures transaction atomicity by recording all database modification in the log.

↳ Assume that transactions execute serially.

↳ Transaction starts by writing <Ti start> record to log.

↳ A write(X) operation results in a log record <Ti, X, V> being written, where V is the new value of X.

Note: old value isnot needed for this scheme.

↳ The write isnot performed on X at this time, but is deferred.

↳ When Ti partially commits <Ti commit> is written to the log.

↳ Finally the log records are read and used to actually execute the previously deferred writes.

↳ Using log, the system handle any failure that results in information loss. The recovery schemes uses

the following recovery procurers:

- Redo (T_i) sets the value of all data items updated by transaction T_i to the new values. New value can be found in the log.

The redo() operation must be idempotent:

- Executing it several times must be equivalent to executing it once.
- This is required if we are to generate correct behaviour even if a failure occurs during the recovery process.

↳ 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.

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

↳ If log on stable storage at time of crash is given above (i.e a, b, c)

* No redo actions need to be taken for case 'a'
* redo(T0) must be performed for case b since <T0 commit> is present.
* redo(T0) must be performed followed by redo(T_1) since <T0 commit> and <T_1 commit> are present.

Immediate Database Modification.

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

↳ Data modification written by active transactions are called uncommitted modifications.

↳ Since undoing may be needed, update logs must have both old value and new value.

↳ Recovery procedure has two operations instead of one:

i. 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 of T_i.

ii. 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 of T_i.

↳ Both operations must be idempotent. i.e even if the operation is executed multiple times the effect is the same as if it executed once

↳ When recovering after failure:

i. Transaction T_i needs to be undone if the log contains the record $<T_i$ start>, but doesnot contain the record $<T_i$ commit>

ii. 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.

Example:

a.

<To start>
<To, A, 1000, 950>
<To, B, 2000, 2050>

Undo(To):
B is restored to 2000 and A to 1000

b. <To start>
<To, A, 1000, 950>
<To, B, 2000, 2050>
<To commit>
<T1 start>
<T1, C, 700, 600>

Undo(T1) and redo(To):
C is restored to 700, and then A and B are set to 950 and 2050 respectively

c. <To start>
<To, A, 1000, 950>
<To, B, 2000, 2050>
<To commit>
<T1 start>
<T1, C, 700, 600>
<T1 commit>

Redo(To) and redo(T1):
A and B are set to 950 and 2050 respectively. Then C is set to 600.

Checkpoints

↳Problems in recovery procedure as discussed earlier:

i. Searching the entire log is time consuming
ii. We might unnecessarily redo transactions which

have already output their updates to the database.

↳ Streamline recovery procedure by periodically performing check pointing.

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

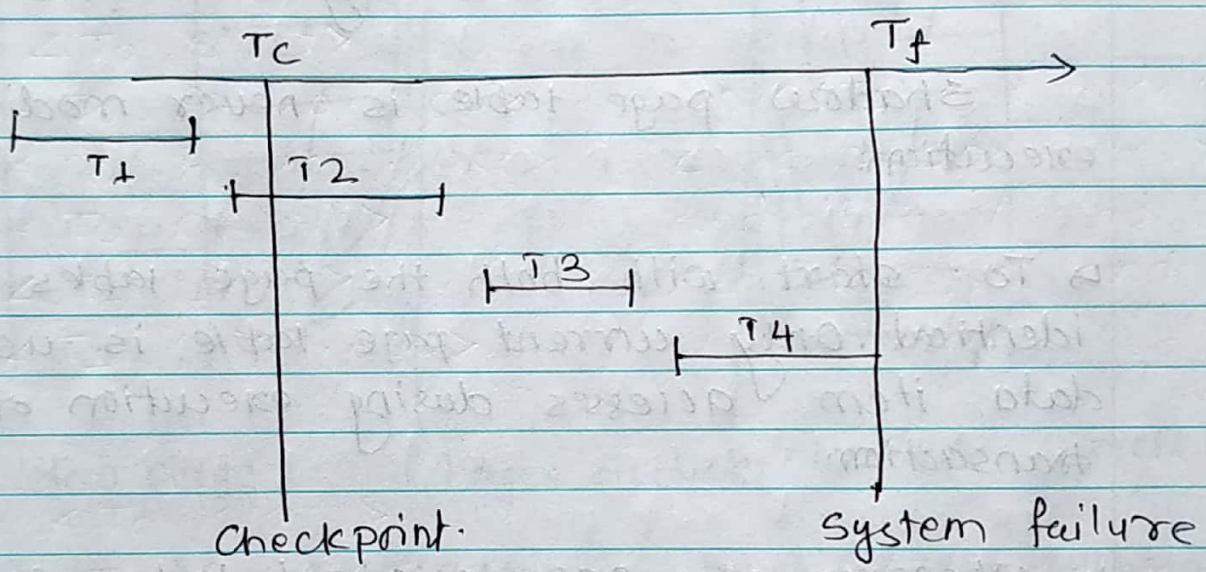
↳ During recovery we need to consider only the most recent transaction T_i that started before the checkpoint
i. Scan backwards from end of log to find the most recent <checkpoint> record.
ii. Continue scanning backwards till a record $<T_i$ start> is found.
iii. Need only consider the part of log following above start record. Earlier part of the log can be ignored during recovery, and can be erased whenever desired.

↳ After identified Transaction T_i, the redo and undo operations to be applied to the T_i and all T_j that started execution after Transaction T_i.

For all transactions (starting from T_i or

later) with no <Ti commit>, execute undo (Ti). (Done only in case of immediate modifications).

↳ Scanning forward in the log, for all transactions starting from Ti or later with a <Ti commit>, execute redo (Ti).



T_1 can be ignored (updates already output to disk due to checkpoint)

T_2 and T_3 redone.

T_4 undone.

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.

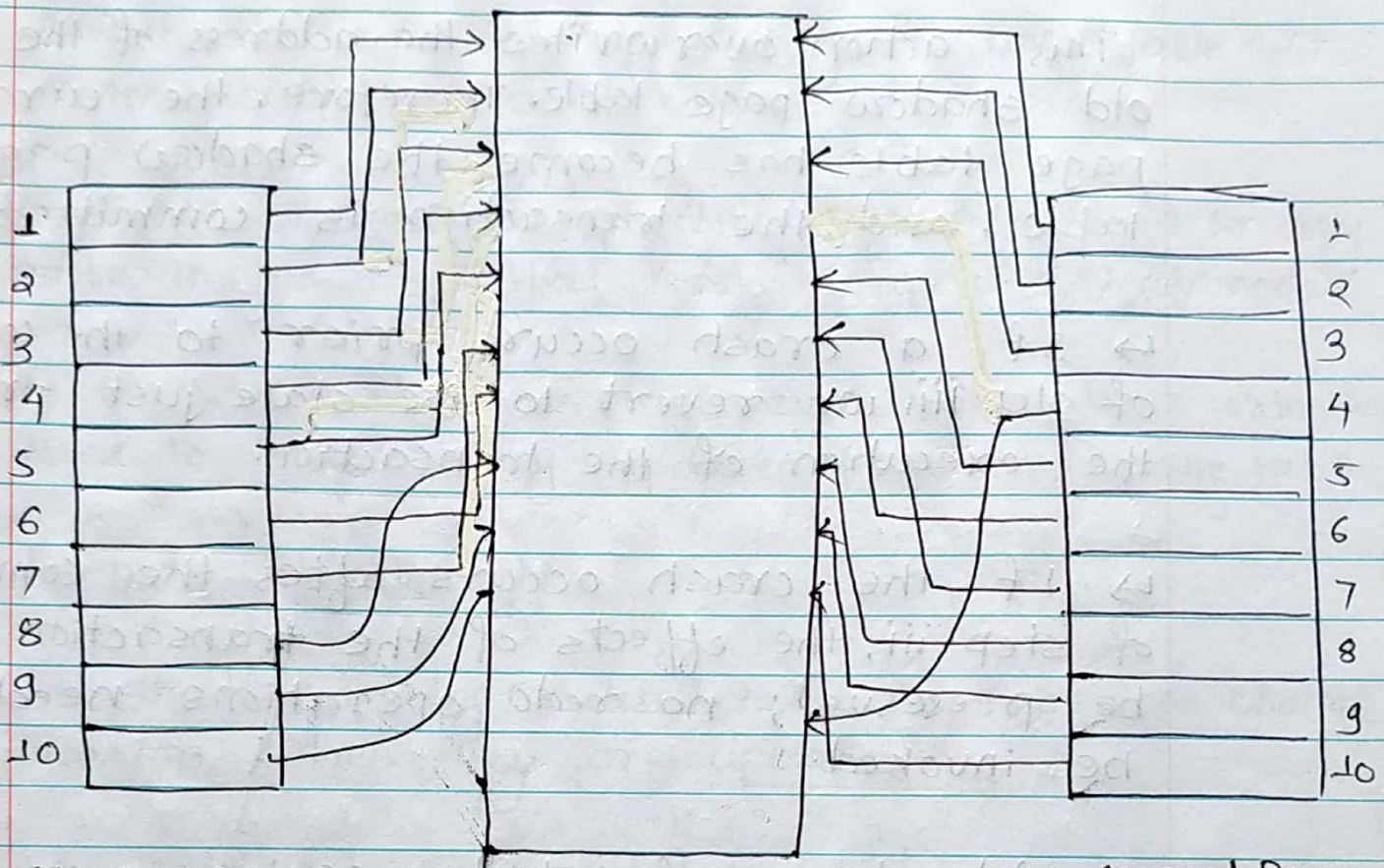
↳ To start with, both the page tables are identical. Only current page table is used for data item accesses during execution of the transaction.

↳ 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.



Shadow page Pages on disk current Page

To commit a transaction, we must do the following:

i. Ensure that all buffer pages in main memory that have been changed by the transactions are output to disk.

ii. Output the current page table to the disk. Note that we mustnot overwrite the shadow page table, since we may need it for recovery from a crash.

iii. Output the disk address of the current page table to the fixed location in stable storage containing the address of the shadow page table.

This action overwrites the address of the old shadow page table. Therefore, the current page table has become the shadow page table, and the transaction is committed

↳ If a crash occurs prior to the completion of step iii, we revert to the state just prior to the execution of the transaction.

↳ If the crash occurs after the completion of step iii, the effects of the transaction will be preserved; no redo operations need to be invoked.

↳ Advantages of shadow-paging over log-based schemes:

i. no overhead of writing log records.
ii. Recovery from crashes is significantly faster (since no undo or redo operations are needed).

Disadvantages:

i. Commit overhead:
Requires multiple blocks to be output - the actual data blocks, the current page table, & the disk address of the current page table. (log based schemas need output only the log records)

can be reduced by using a page table structured like a B+ tree.

No need to copy entire tree, only need to copy paths in the tree that lead to updated leaf nodes.

↳ Commit overhead is high even with above extension. Need to flush every updated page, and page table.

ii. Data Fragmentation:

↳ Shadow paging cause database pages to change location when they are updated.

↳ Data gets fragmented. (related pages get separated on disk)

iii. Garbage Collection:

↳ After every transaction completion, the database pages containing old versions of modified data need to be garbage collected.

↳ Hard to extend algorithm to allow transactions to run concurrently.

↳ Easier to extend log based schemes.

Advanced Recovery Algorithm.

↳ Support for high-concurrency locking techniques. such as those used for B+ tree concurrency control, which release locks early

* supports "logical undo"

↳ Recovery based on "repeating history", whereby recovery executes exactly the same actions as normal processing.

* including redo of log records of incomplete transactions, followed by subsequent undo.

* key benefits
  - ✡ supports logical undo
  - ✡ easier to understand / show correctness.

Logical Undo Logging.

↳ Operations like B+ tree insertions and deletions release locks early.

↳ They cannot be undone by restoring old values (Physical undo), since once a lock is released, other transactions may have updated the B+ tree.

↳ Instead, insertions (resp. deletions) are undone

by executing a deletion (resp. insestion) opesation (known as logical undo)

↳ In logical undo method; a separate undo log file is created along with log file.

↳ In undo file, for any insertion operation, respective deletion operation will be mentioned to rollback changes.

↳ Similarly for each deletion operation, respective insestion opesation will be descsibed.

↳ For example: suppose a transaction T_1 is adding $x = x+5$. Then in our physical logging method, we will have log like $<T_1, X, 10, 15>$ indicating x value is changed from 10 to 15.

↳ In case of failure, we know what the previous value of x was and we can easily undo x to 10.

↳ But it will not work in case of B+ trees. We will have to maintain how to undo x to 10 i.e; a separate logical undo file is created where we will mention undo for $x = x+5$ as $x = x-5$.

↳ Suppose we have insested a new entry for student as 'INSERT INTO STUDENT VALUES (200, ...'). The logical undo ~~opes~~ file will contain

undo operation for this as 'DELETE FROM STUDENT WHERE STD_ID = 200'

↳ Redo for the transaction can be done by following the log file - physical log. We will not maintain logical log for redoing the transaction.

↳ This is because; the state of the record would have changes by the time system is recovered. Some other transactions would have already executed and will lead to logical redo log to be wrong. Hence the physical log itself is re-executed to redo the operations.

Operation Logging.

↳ In operation logging, apart from physical undo and redo logs, we will have logical undo logs too. Each one of them is useful and is used depending on when the crash has occured.

↳ Let T_i be the transaction and O_j be the operation in T_i. Let U be the logical undo information. Then operation logging for an operation in a transaction is done as follows:

i. When an operation begins in the transaction, an operation log $< T_i, O_j, \text{operation-begin} >$ is logged. It indicate the beginning of opesation.

ii. When the operation is executed, logs for them are insested as any other normal logging method. It will contain physiceal undo and redo informations.

iii. When the operation is complete, it will log <Ti, Oj, operation_end, U>. This will have logical undo information for reverting the changes.

...
<Ti start>
<Tj, Oi, Operation-begin>
<Ti, vasiable, old_value, New_value>
<Ti, Oi, Operation_end, U>
...
<Ti, End>

Suppose we have to insert values for (X, Y) as ('ABC', 20). Then opesation log for this will be as follows:

When abort or crash occurs in the system while transaction is executing:

↳ If it crashes before operation_end, then the physical undo information is used to revert the operation. Here log willnot have opesation_end; henie system will automatically take physical undo from the log i.e X and Y will be reverted to its old values.

↳ If the system crashes aftes operation_end, then

physical undo is ignored and logical undo is used to revest the changes. i.e. DELETE is used to revert the changes and it will delete newly added information.

↳ In both the cases above, physical redo information is used to re-excute the changes i.e., values of X and Y are updated to ('ABC', 20)

<T_1, start>
<T_1, O_1, Operation-begin>
<T_1, X, 'MNO', 'ABC'> } Physical undo and Redo steps.
<T_1, Y, 100, 20>
<T_1, O_1, Operation-end, DELETE ('ABC', 20>
<T_1, End>

Transaction Rollback.

↳ When a system crashes while performing the transaction, log entries are used to recover from failuse.

↳ Wheneves there is a failure, the log files will be updated with logs to perform the undo and redo using the already entesed information. i.e., if undo of <T_1, X, 'MNO', 'ABC'> has to be done then it will entes the another log aftes tha crash as <T_1, X, 'MNO'>

Whenever there is crash and system is trying to recover by rolling back, it will scan the logs in reverse order and log entries are updated as below:

↳ If there is log entry <Ti, Variable, old_value, New_value>, then enter undo log as <Ti, Variable, old_value>. This undo log entry is known as redo-only log entry. While recovering, if it finds redo-only record, it ignores it.

↳ If it finds <Ti, Oj, operation_end, U> while traversing log, then rollback the operation using logical undo, U. This logical undo operation is also logged into log file as normal operation execution, but at the end instead of <Ti, Oj, operation_end, U>, <Ti, Oj, Operation-Abort> is logged. Then skip all the operations till <Ti, Oj, operation-begin> is reached. i.e.; it performs all the logical undo operation like any other normal operation and its logs are entered into log file, and all the physical undo operations are ignored.

↳ Let us consider the transaction as below. We can observe that T_1 has two operations O_1 and O_2, where O_1 is completed fully and while performing O_2, system crashes. While recovering it starts scan in reverse from the point where it failed & starts entering the logs for recovering. Hence it finds only <T_1, z, 'abc', 'xyz'> entry in the log while

recovering, and redo-only entry $<T_1, Z, 'abc'>$ for O_2 is entered. Then it finds operation end for O_1. Hence it uses logical undo to rollback the changes by O_1. Though it finds logical undo as 'DELETE', it starts insesting the redo logs for performing 'DELETE'.

↳ This redo logs for delete will in turn delete the changes done by the operation O_1. It then traverses back the physical redo of O_1 without executing it (ignores it) till it reaches $<T_1, start>$, and stops. It adds $<T_1, start>$ to the log file to indicate the end of reverting transaction T_1. We can see this in below log file - after logical undo of O_1, we donot have any logs of physical undo or redo, it jumps to Abort log entries.

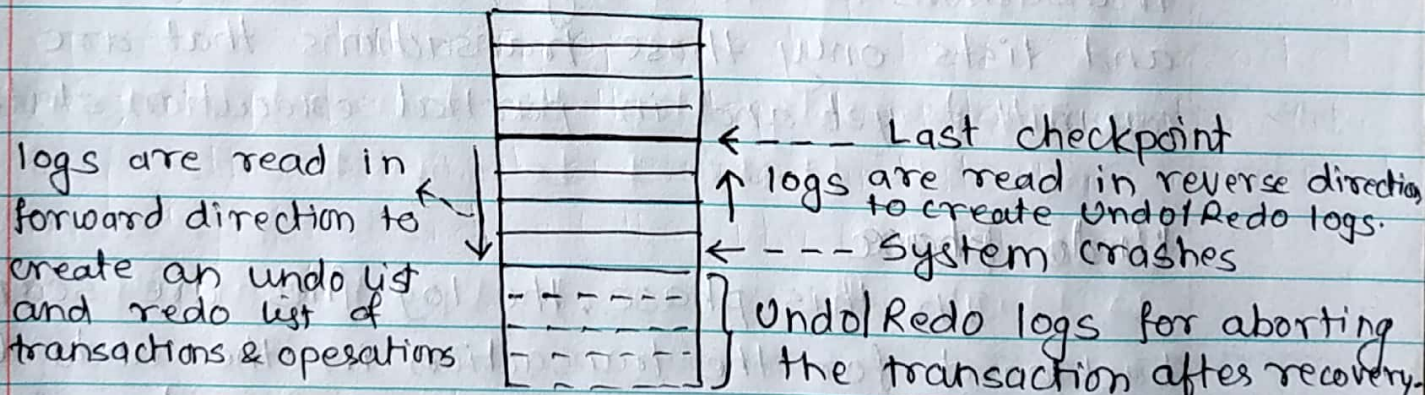
$<T_1, Start>$
$<T_1, O_1, Operation-begin>$
$<T_1, X, 'MNO', 'ABC'>$ } Physical undo & Redo steps
$<T_1, Y, 100, 20>$ }
$<T_1, O_1, operation-end, DELETE (I5, 'ABC', 20)>$
$<T_1, O_2, Operation\_begin>$
$<T_1, Z, 'abc', 'xyz'>$
$<T_1, Z, 'abc'>$ ← Redo-only log is entered for O_2
$<T_1, Y, 20, delete>$ } Logical undo for O_1 entering
$<T_1, X, 'ABC', delete>$ } the redo logs for it
$<T_1, O_1, Operation-Abort>$
$<T_1, Abort>$

Crash Recovery:

Whenever there is a system crash, the transactions which were in execution phase has to be recovered and DB has to be brought to consistent state. Log files are checked to do redo and undo operations. It has two phases.

Redo phase:

Though the transactions and operations are rolled back in reverse order of log file entries, the recovery system maintains the recovery log list for undoing and redoing the operations by scanning the logs from the last checkpoint to the end of file.



That means, undo/redo logs will have list of operations and how to execute them, and are entered into the log file itself. A separate list of entries will be created for maintaining the list of transactions/operations which needs to be undone while recovering. This will be created by scanning the log files from last checkpoint

to the end of the file (forward direction). While creating the undo list, all other operations which are not part of undo list are redone.

While performing the forward scan to create undo list, L, it checks if

<Ti, start> found, then adds Ti to undo list L
<Ti, commit> or <Ti, Abort> is found, then it deletes Ti entry from undo list, L

Hence undo list will have all the transactions which are partially performed and all other committed transactions are re-done (redoing the transaction is not exactly as re-executing them. This forward scanning assumes that those transactions are already performed and committed, and lists only those transactions that are not committed yet and in partial execution state.)

Undo phase

In this phase, the log files are scanned backward for the transaction in the undo list Undoing of transactions are performed as described in transaction rollback. It checks for the end log for each operations, if found then it performs logical undo, else physical undo by entesing the logs in log files.

This is how transaction is redone and undone

to maintain the consistency and atomicity of the transaction

Check pointing:

↳ Check pointing is the mechanism to mark the entries in the log file that those changes are permanently updated into database, and if there is any failure, log files need not be traversed beyond that point.

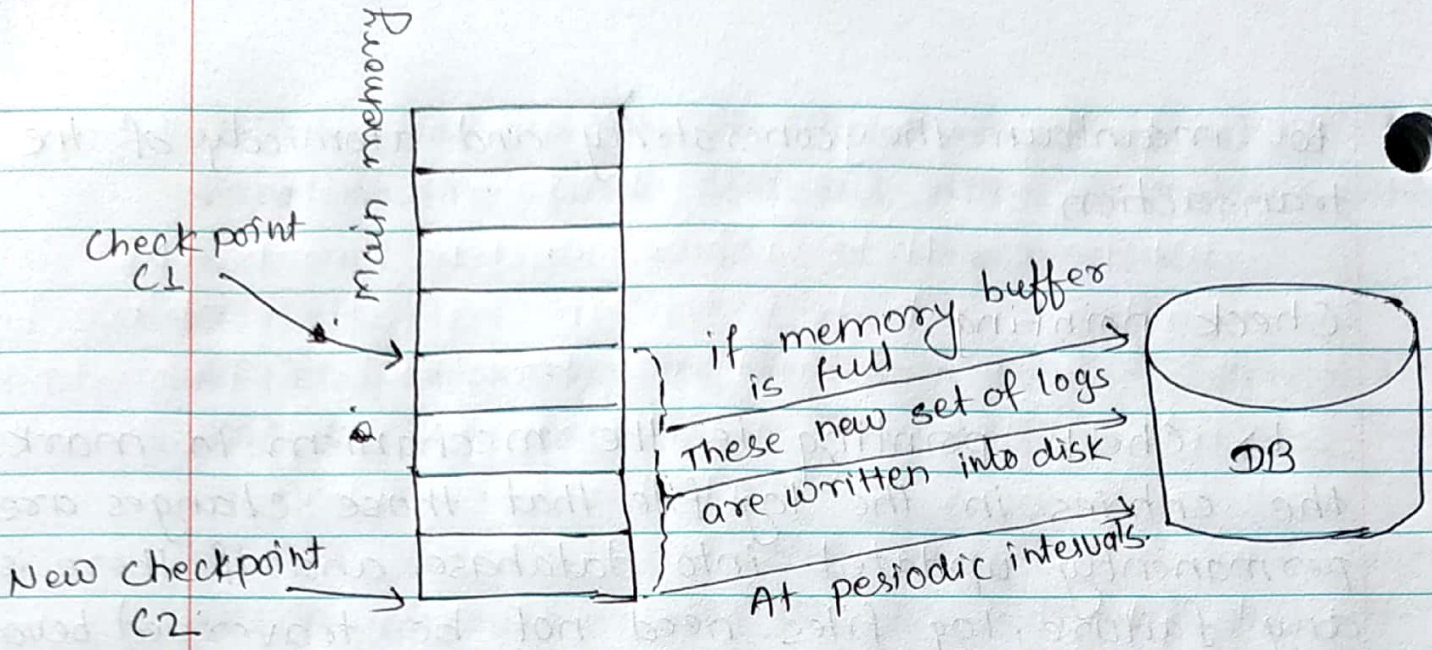
↳ Only those entries after checkpoint are not written to DB, and have to be redone/undone. This is done at periodic intervals or as per the schedule.

↳ Checkpointing is done as follows:

i. It checks for the log records after the last checkpoint and outputs it to the stable memory/disks.

ii. If the buffer blocks in main memory is full, then it outputs the logs into disks.

iii. If there is any new checkpoint is defined, then all the entries from last checkpoint to the new check points are written to disks. But any transactions will not get executed during this check pointing process.



Fuzzy Checkpointing:

Fuzzy checkpointing, in contrast to normal check pointing allows transactions to execute while logs are being copied to disk. During fuzzy check pointing it follows below steps:

↳ Temporarily stops the transactions to make note of blocks to be copied.

↳ Marks the new checkpoint L in the log.

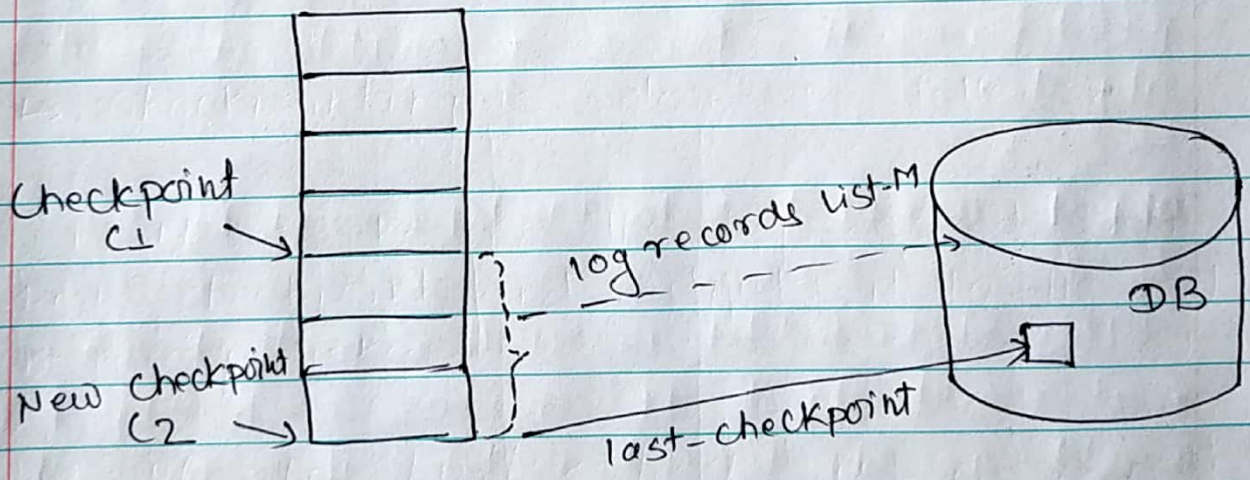
↳ Creates a list M for all the logs between the last checkpoint to new checkpoint. i.e; M is the list of log records which are yet to be written to disk

↳ Once all M is listed, it allows the transaction to execute. Now the transaction will start

entering the logs after the ~~logs~~ new checkpoint L. It shouldnot enter the logs into the blocks that are in M or old checkpoints.

↳ The buffer blocks in list M are written to the disk or stable storage. No transactions should update these blocks. In addition, all the records in these blocks in list M are written to the disk first, and then the block is updated to the disk.

↳ Disk should have pointer to the last checkpoint. Last-checkpoint in the main memory at fixed location. This will help to read the blocks for the next update and maintain new list M.



Whenever there is recovery from failure, then logs are read from the last_checkpoint stored in DB. This is because logs before last-checkpoint are already updated to DB and those after these points have to be written. These logs are recovered as described in above methods.