CRASH RECOVERY

Failure Classification

A transaction has to abort when it fails to execute or when it reaches a point from where it can't go any further.

Reasons for a transaction failure can be :

- **Logical Error :** Where a transaction can't complete because it has some code error or any internal error condition. Example : bad input, overflow error, data not found error.
- **System Error :** Where database itself terminates an active transaction because the DBMS is not able to execute it or it has to stop because of some system condition.
  Example : In case of deadlock or resource unavailability, the system aborts an active transaction.

System Crash

There are problems − external to the system − that may cause the system to stop abruptly and cause the system to crash. For example, interruptions in power supply may cause the failure of underlying hardware or software failure.

Examples may include operating system errors.

Disk Failure

In early days of technology evolution, it was a common problem where hard-disk drives or storage drives used to fail frequently.

Disk failures include formation of bad sectors, unreachability to the disk, disk head crash or any other failure, which destroys all or a part of disk storage.

Recovery And Atomicity

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 −

- It should check the states of all the transactions, which were being executed.
- A transaction may be in the middle of some operation; the DBMS must ensure the atomicity of the transaction in this case.
- It should check whether the transaction can be completed now or it needs to be rolled back.
- No transactions would be allowed to leave the DBMS in an inconsistent state.

There are two types of techniques, which can help a DBMS in recovering as well as maintaining the atomicity of a transaction −

- Maintaining the logs of each transaction, and writing them onto some stable storage before actually modifying the database.

- Maintaining shadow paging, where the changes are done on a volatile memory, and later, the actual database is updated.

Log Based Recovery

Log is a sequence of records which maintain the records of actions performed by a transaction. It is important that the logs are written prior to the actual modification & stored on a stable storage media. (Log is the most commonly used structure for recording database modification.)
Updated log has the following fields :

- Transaction identifier
- Data item identifier
- Old value (Prior to write)
- New value (After write)

Example of log record

< T_1 Start >
< T_1, X_2, 10, 15 >
< T_1 Commit >
< T_1 Abort >

Here T_1 = Transaction identifier
X_2 = Data item
10 = Old value
15 = New value

If T_1 commits then X_2 = new value = 15.
If T_1 aborts then X_2 = old value = 10 & result roll back.

Log Based Recovery works as follows : -

- The log file is kept on a stable storage media.
- When a transaction enters the system and starts execution, it writes a log about it.
  < T_n Start > => Transaction identifier
- When the transaction modifies an item X, it writes log as
  < T_n, X, V_1, V_2 >
- It reads T_n has changed the value X, from V_1 to V_2.
  When the transaction finishes , it logs : < T_n Commit >

Write Ahead Log Strategy

Log is written before any update is made to the database. Transaction is not allowed to modify the physical database until the UNDO portion log is written to stable storage.

Operation in Recovery Procedure

a) UNDO (T_i) : Restore Old Value
b) REDO (T_i) : Updates By New Value

Approaches in transaction

Recovery Procedure

1. Deferred Database Modification

- Transaction operation don't immediately update the physical database.
- Only Transaction log is updated.
- Database is physically updated only after the transaction reaches its commit point.

No-UNDO/REDO Algorithm

Example

Log :- $< T_1$ Start >

$< T_1$, A, 2, 3 >

$< T_1$ Commit >

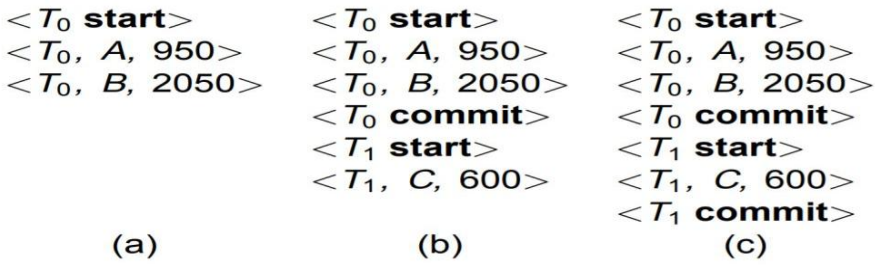
Here after commit operation only, Transaction will update the value of A from 2 to 3.

This scheme ensures atomicity despite failures by recording all modifications to log, but deferring all the writes to after partial commit

- 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 for X. The write is not performed on X at this time, but is deferred.
- When Ti partially commits, < Ti commit > is written to the log
- Finally, log records are used to actually execute the previously deferred writes.
- During recovery after a crash, a transaction needs to be redone if and only if both < Ti start > and < Ti commit > are there in the log.
- Redoing a transaction Ti (redo(Ti)) 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
- example transactions T0 and T1 (T0 executes before T1):

T0:		T1: read(C)
	A := A − 50	C := C − 100
	write(A)	write(C)
	read(B)	
	B := B + 50	
	write(B)	

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



- If log on stable storage at time of crash is as in case:
  - No redo actions need to be taken
  - redo(T0) must be performed since is present
  - redo(T0) must be performed followed by redo(T1) since and are present

2. Immediate Database Modification

- Database is immediately updated by the transaction operation during the execution of transaction even before it reaches commit point.
- In case of T_r abort/failed before it reaches commit point, a Rollback or UNDO operation needs to be done to restore the database to its consistent state.

UNDO/No-REDO ALgorithm

Example

Log :- < T_1 Start >
< T_1, A, 2, 3 >
< T_1 Commit >

Here at second stage, transaction will modify value of A from 2 to 3 after that it will commit.

This 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. 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.

Example

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

Recovery procedure has two operations instead of one :

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

When recovering after failure:

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

Undo operations are performed first, then redo operations

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

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

Recovery actions in each case above are:

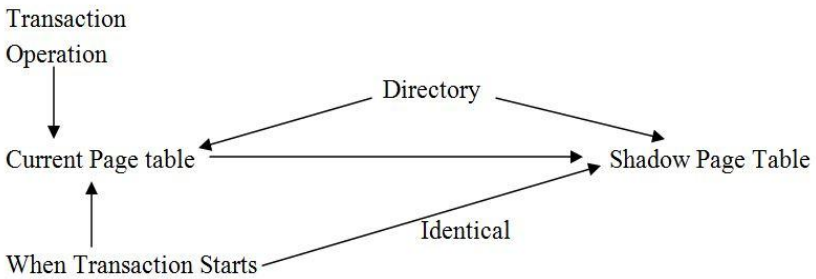
a) undo(T0): B is restored to 2000 and A to 1000.

b) undo(T1) and redo(T0): C is restored to 700, and then A and B are set to 950 and 2050 respectively.

c) redo(T0) and redo(T1): A and B are set to 950 and 2050 respectively. Then C is set to 600.

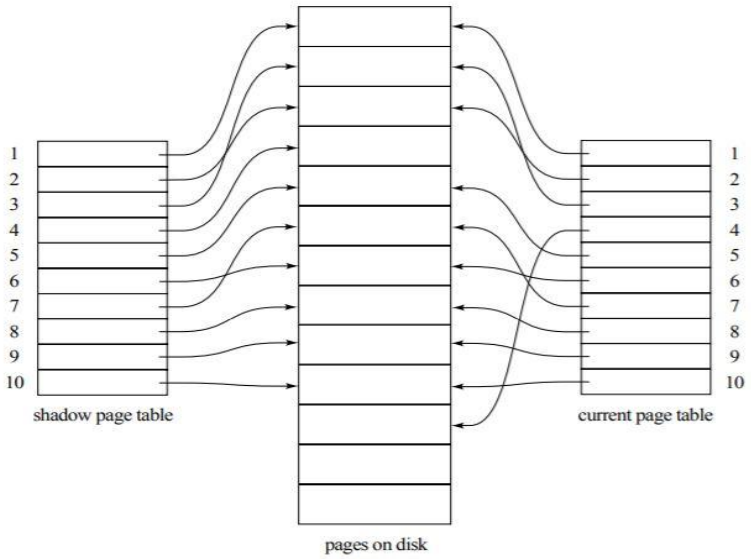
Shadow ept of Saptial Database

- An alternative approach to log based recovery : it is useful if transactions execute serially.
- Maintains two page tables during the life cycle of transaction
- When transaction starts both page tables are identical
- Shadow page table is never changed over duration of transaction
- Current page table may changed during write operation
- All input and output operations use the current page table to locate database pages on disk
- Store shadow page table in non-volatile storage



[When transaction commits, system writes current page table to non-volatile storage. the current page table then become new shadow page table.

Example of shadow paging



Shadow and current page tables after write to page 4.

To commit a transaction:

- Flush all modified pages in main memory to disk
- Output current page table to disk
- Make the current page the new shadow page table
  - keep a pointer to the shadow page table at a fixed (known) location on disk.
  - to make the current page table the new shadow page table, 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).

Advantages of shadow paging over log based recovery

- Log record overhead is removed i.e. we don't have to maintain logs.
- Faster recovery because we remove page table directly. we don't have to do any REDO/UNDO operations.

Disadvantages

- Commit overhead is high (many pages need to be flushed)
- Data gets fragmented (related pages get separated)
- After every transaction completion, the database pages containing old versions of modified data need to be garbage collected and put into the list of unused pages
- Hard to extend algorithm to allow transactions to run concurrently

Advanced Recovery Techniques

- Support high-concurrency locking techniques, such as those used for B+-tree concurrency control
- 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. insertion) operation (known as logical undo).
- For such operations, undo log records should contain the undo operation to be executed; called logical undo logging, in contrast to physical undo logging.
- Redo information is logged physically (that is, new value for each write) even for such operations.

Operation logging is done as follows:

- When operation starts, log < Ti, Oj, operation-begin>. Here Oj is a unique identifier of the operation instance.
- While operation is executing, normal log records with physical redo and physical undo information are logged.
- When operation completes, is logged, where U contains information needed to perform a logical undo information.
- If crash/rollback occurs before operation completes, the operation-end log record is not found, and the physical undo information is used to undo operation.
- If crash/rollback occurs after the operation completes, the operation-end log record is found, and logical undo is performed using U; the physical undo information for the operation is ignored.

Rollback of transaction Ti is done as follows:

- Scan the log backwards
    1. If a log record is found, perform the undo and log a special redo-only record < Ti, X, V1 >.
    2. If a < Ti, Oj, operation-end, U > record is found
        - Rollback the operation logically using the undo information U. Updates performed during roll back are logged just like during normal operation execution.
        - At the end of the operation rollback, instead of logging an operation-end record, generate a record < Ti, Oj, operation-abort>.
        - Skip all preceding log records for Ti until the record < Ti, Oj, operation-begin> is found
    3. If a redo-only record is found ignore it
    4. If a < Ti, Oj, operation-abort> record is found, skip all preceding log records for Ti until the record < Ti, Oj, operation-begin> is found.
- Stop the scan when the record < Ti, start> is found

- Add a < Ti, abort> record to the log

Some points to note:

- Cases 3 and 4 above can occur only if the database crashes while a transaction is being rolled back.
- Skipping of log records as in case 4 is important to prevent multiple rollback of the same operation.

The following actions are taken when recovering from system crash

1. Repeat history by physically redoing all updates of all transactions, scanning log forward from last < checkpoint L> record
   - undo-list is set to L initially
   - Whenever is found Ti is added to undo-list
   - Whenever or is found, Ti is deleted from undo-list This brings database to state as of crash, with committed as well as uncommitted transactions having been redone. Now undo-list contains transactions that are incomplete, that is, have neither committed nor been fully rolled back.

2. Scan log backwards, performing undo on log records of transactions found in undo-list. Transactions are rolled back as described earlier.
   - When is found for a transaction Ti in undo-list, write a log record.
   - Stop scan when records have been found for all Ti in undo-list

This undoes the effects of incomplete transactions (those with neither commit nor abort log records). Recovery is now complete.

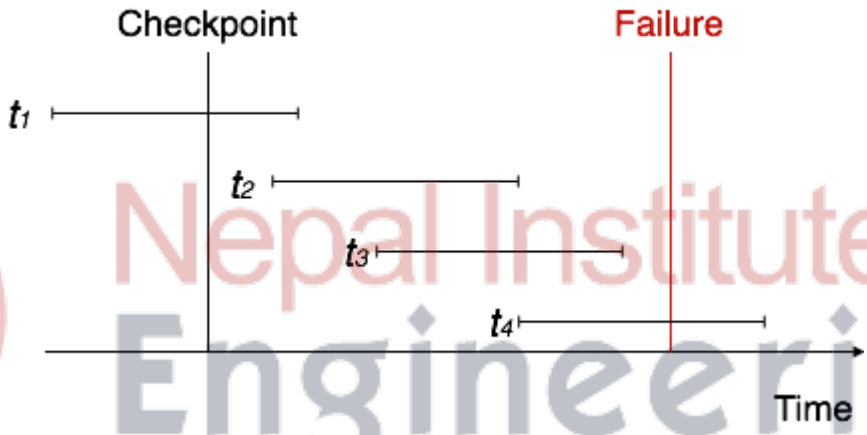
Checkpoint Recovery

When more than one transaction are being executed in parallel, the logs are interleaved. At the time of recovery, it would become hard for the recovery system to backtrack all logs, and then start recovering. To ease this situation, most modern DBMS use the concept of 'checkpoints'.

Keeping and maintaining logs in real time and in real environment may fill out all the memory space available in the system. As time passes, the log file may grow too big to be handled at all. Checkpoint is a mechanism where all the previous logs are removed from the system and stored permanently in a storage disk. Checkpoint declares a point before which the DBMS was in consistent state, and all the transactions were committed.

Recovery

When a system with concurrent transactions crashes and recovers, it behaves in the following manner −



- The recovery system reads the logs backwards from the end to the last checkpoint.
- It maintains two lists, an undo-list and a redo-list.
- If the recovery system sees a log with <T_n, Start> and <T_n, Commit> or just <T_n, Commit>, it puts the transaction in the redo-list.
- If the recovery system sees a log with <T_n, Start> but no commit or abort log found, it puts the transaction in undo-list.

All the transactions in the undo-list are then undone and their logs are removed. All the transactions in the redo-list and their previous logs are removed and then redone before saving their logs.

Q. Consider following log contents when a crash occurs. Briefly explain how a recovery would be done.

<T_0 Start>
<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>
<T_0, Commit>
<T_1 Start>
<T_1, C, 700, 600>

Crash occurs here

Solution : Recovery Actions

- REDO (T_0) & UNDO (T_1) : -
  - A and B are set to 950 and 2050 and C is restored to 700.
  - Log records < T_1, C, 700 > and < T_1 Abort > are written.

Q. Consider the following log contents

< T_0 Start >
< T_0, A, 1000, 950 >
< T_0, B, 2000, 2050 >

Actions for recovery

- UNDO (T_0) - B is restored to 2000 and A to 1000. And log records < T_0, B, 2000 > , < T_1, A, 1000 > and < T_0 Abort > are written out.

Explain this recovery actions briefly.

Q. Consider the following log contents

< T_0 Start >
< T_0, A, 1000, 950 >
< T_0, B, 2000, 2050 >
< T_0 Commit >
< T_1 Start >
< T_1, C, 700, 600 >
< T_1 Commit >
Crash occurs here

Recovery Actions :-

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

Explain this recovery actions briefly.

* Suppose following contents are present in the log when a crash occurs. Explain what happens for a log based recovery?*

< T_0 Start >
< T_0, B, 2000, 2050 >
< T_1 Start >
Checkpoint {T_0 , T_1}
< T_1, C, 700, 600 >
< T_1 Commit >
< T_2 Start >
< T_2, A, 500, 400 >
< T_2 Abort >
← End of log at crash and T_2 is incomplete at crash
< T_2, A, 500 > } T_2 is rolled back in UNDO pass
< T_2 Abort > } Log records added during recovery

Nepal Institute of Engineering

By : Er. Deepak Kumar Singh