5. Recovery with Concurrent Transactions (Rollback, Checkpoints, Commit)

Rollback

A rollback is an operation which returns the database to some previous state. Rollbacks are important for database integrity, because they mean that the database can be restored to a clean copy even after erroneous operations are performed.

1

Apr. 2018 – *5M*
Explain the recovery with concurrent transaction.

They are crucial for recovering from database server crashes; by rolling back any transaction which was active at the time of the crash, the database is restored to a consistent state.

The rollback feature is usually implemented with a transaction log, but can also be implemented via multiversion concurrency control.

Checkpoints

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

1

Apr. 2018 – *3M*
Write a short note on checkpoints.

Commit

A transaction T reaches its commit point when all its operations that access the database have been executed successfully and the effect of all the transaction operations on the database have been recorded in the log.

After the commit point, the transaction is said to be committed and its effect is assumed to be permanently recorded in the database. The transaction then writes an entry [commit, T_i] into the log. If a system failure occurs, the log is searched back for all transactions T that have written a [start–transaction, T] entry in the log, but no [commit, T] entry yet; such transaction may have to be rolled back to undo their effect on the database.

Transactions that have written their commit entry in the log must also have recorded all their write entries in the log. So, their effect on the database can be redone from the log entries.

5.1 Recovery Procedure after System Crash

When the system recovers from a crash, the recovery manager component takes control of the systems. The recovery manager component constructs two lists: the Undo list and the Redo list. The Undo list consists of transactions that need to be redone. The two lists are constructed as follows:

1. Initially both the lists are empty.
2. The recovery manager scans the log backward, examining each record, until it finds the first <checkpoint> record.
   a. For each commit record found , i.e. $<T_i$, commit> , T_i is added to the redo-list.
   b. For each start record, i.e $<T_i$, start>, T_i is added to the undo-list, if its not found in the redo-list.

During the regular processing the checkpoint record is inserted into the log. The format of a checkpoint record is <checkpoint L>, where L is the List of transactions active at the time of checkpoint.

Once both the undo and the redo lists are constructed, the recovery manager checks the list L in the checkpoint record. For each transactions T_i in L, if T_i is not in redo-list, then it adds T_i to the undo-list.

Once the final redo and undo lists have been constructed, the recovery procedure proceeds as follows:

1. The recovery manager rescans the log from the most recent record backward, and does an undo for each log record that belongs to transaction T_i in the undo list. The log records of the transactions from the redo-list are ignored in this step. The scan stops when the $<T_i$, start> records have been found for every transactions T_i in the undo list.
2. The system finds the most recent <checkpoint L > record on the log. The step may involve scanning the log forward, in case the checkpoint record was passed in step 1.
3. The recovery manager scans the log forward from the most recent <checkpoint L> record and performs a redo for each log record that belongs to a transaction T_i that is on the redo list. The log records of transactions in the undo list are ignored in this phase.

The log is processed or scanned backwards in the 1st step, since the corrections of the database state has to be ensured. After all the transactions in the undo list have been undone, it redoes the actions of the transactions in the redo-list.

When the recovery process has completed, the normal transaction processing resumes. It's necessary that all undo operations have to be performed, before the redo operations. This is depicted in the following *example:*

Consider a data item X, that has an initial value = 10. Suppose that a transaction T_i updates the data item A to 20 and it aborts, T_i is rolled back, thus restarting A to the value 10. Suppose that another transaction T_j updates A to 40 and commits, after which the System Crashes. The state of the log at the time of crash is

$<T_i, X, 10, 20>$

$<T_j, X, 10, 30>$

$<T_j, commit>$

If redo phase is performed first, X will be set to 30, then in the undo pass, X will be set to 10. Then the final value of X would be 10, which is wrong, since the final value of X should be 30, which ensures the durability property of transactions.

6. Log Based Recovery Techniques (Deferred and Immediate Update)

3

Apr. 2018 – *1M*
What are various entries in system log?

Apr. 2018 – *4M*
Write a short note on log based recovery.

Oct. 2017 – *1M*
What is log?

The System Log

To be able to recover from failures that affect transactions, the system maintains a log to keep track of all operations that affect the values of database items. This information may be needed to permit recovery from failures.

The typical entries called log records are:

1. **[start – transaction, T]**
   Indicates that transaction T has started execution.
2. **[write – item, T, X, old – value, new-value]**
   Indicates that transaction T has changed the value of database item X from old – value to new-value.

3. **[commit , T]**

   Indicates that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.

4. **[abort, T]**

   Indicates that transaction T has been aborted.

Log Based Recovery

Log based recovery is one of the schemes to achieve the recovery from transaction failures. There are two techniques for using log, to achieve the recovery and ensure atomicity in case of failures.

i. Deferred update

ii. Immediate update

The *deferred technique* ensures transaction atomicity by recording all database modification in the log, but deferring the execution of all write operations of transaction until the transaction partially commits.

The *immediate update technique* allows database modifications to be output to the database while the transaction is still in active state. Data modifications written by active transactions are called *uncommitted modifications*. If a failure occurs during execution, the system must use the old value field of log records.

6.1 Deferred Update

1

Oct. 2017 – *3M*
Explain deferred database modification.

The idea behind deferred update technique is to postpone or defer any actual updates to the database itself until the transaction completes its execution successfully and reaches its commit point.

During the execution of the transactions, the updates done by the transactions are recorded in the log and in the transaction workspace. After the transaction reaches its commit point and the log is force written to the disk, the updates are recorded in the database itself.

[*Force-writing: Before the transaction reaches its commit point, any portion of the log that has not been written to the disk yet must now be written to the disk. This process is called force-writing the log file before committing a transaction.*]

If a transaction fails before reaching its commit point, there is no need to undo any of its operations, because the transaction has not affected the database in any way.

Hence, this technique is also called no *undo/redo algorithm*. The redo is needed in case the system fails after the transaction commits but before all its changes are recorded in the database. In this case, the transaction operations are redone from the log entries.

Examples

1. The transaction T_0 transfers Rs.50/- from account A to account B. Original values of A and B are Rs.1000 and Rs.2000 respectively.

T_0:	read(A)
	A := A – 50
	write(A)
	read(B)
	B := B + 50
	write(B)

Consider a second transaction T_1 that withdraws Rs.100 from 'C',

T_1:	read(C)
	C := C – 100
	write(C)

These two transactions are executed serially, $<T_0, T_1>$. The log containing information of these two transactions are:

$<T_0$, start$>$

$<T_0$, A, 950$>$

$<T_0$, B, 2050$>$

$<T_0$, commit$>$

$<T_1$, start$>$

$<T_1$, C, 600$>$

$<T_1$, commit$>$

The actual output can take place to database in various orders. One such order is given below:

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

If system crashes before the completion of transactions:

Case 1

Crash occurs just after the log record for write (B) operation

Log contents: $<T_0$, start$>$

$<T_0$, A, 950$>$

$<T_0$, B, 2050$>$

$<T_0$, commit$>$ is not written, hence no redo is done. The transaction has to be resubmitted again.

Case 2

Crash occurs after the log record for write (C).

Log contents: $<T_0$, start$>$

$<T_0$, A, 950$>$

$<T_0$, B, 2050$>$

$<T_0$, commit$>$

$<T_1$, start$>$

$<T_1$, C, 600$>$

Here, $<T_0$, start$>$ and $<T_0$, commit$>$ are found, so T_0 is redone, where as T_1 is resubmitted again.

Case 3

Crash occurs just after the log record $<T_1$, commit$>$

Log contents: <T_0, start>

<T_0, A, 950>

<T_0, B, 2050>

<T_0, commit>

< T_1, start>

<T_1, C, 600>

<T_1, commit>

Here both T_0 and T_1 have their start and commit hence both will be redone.

2. Consider 4 transactions as under:

T_1	T_2	T_3	T_4
read(A)	read(B)	read(A)	read(B)
read(D)	write(B)	write(A)	write(B)
write(D)	read(D)	read(C)	read(A)
	write(D)	write(C)	write(A)

The system log at the point of crash.

<T_1, start>

<T_1, D, 20>

<T_1, commit>

<check point>

<T_4, start>

<T_4, B, 15>

<T_4, A, 20>

<commit T_4>

< T_2, start>

<T_2, B, 12>

<T_3, start>

<T_3, A, 30>

<T_2, D, 25> ← system crash.

Looking into the above *example,* T_2 and T_3 are ignored as they have to be resubmitted.

T_4 is redone because its commit point is after the last system checkpoint.

Transaction Roll-Back

In case of a system crash

1. Because there can be some updates of the committed transactions that are not written into the database, effects of the committed transactions are redone to the database.
2. Only the last updates of each item have to be redone.

6.2 Immediate Update Method for Recovery

It allows the database modifications to be output to the database while transaction is still in active state. Database modifications written by active transactions are called *uncommitted modifications*.

Execution of transaction proceeds is as follows

- Before T_i starts its execution, the record <T_i ,start> is written to the log.
- Before executing any Write (x), i.e., before modifying the database for write operation, it writes an update record <T_i, x, oldvalue, newvalue> to the log.
- When T_i partially commits, the record <T_i, commit> is written to log.

As an illustration consider the same *example* of bank accounts and transactions T_0 and T_1.

The log is as follows

<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>

The order in which output took place to both database system and log as a result of execution of T_0 and T_1, is:

<T_0, start>
<T_0, A, 1000, 950> → A = 950
<T_0, B, 2000, 2050> → B = 2050
<T_0, commit>
<T_1, start>
<T_1, C, 700, 600> → C = 600
<T_1, commit>

The procedure for this recovery scheme is as follows

i. Use two lists of transactions maintained by the system.

The committed transactions since the last checkpoint and the active transactions.

ii. UNDO all the write operations of the active (uncommitted) transactions. The operations should be undone in the reverse of the order in which they were written into the log.

iii. REDO all the write operations of the committed transactions from the log, in the order in which they were written into the log.

Consider the following conditions of failure for transactions T_0 and T_1:

Case 1

Failure occurs just after the log record for write (B) operation.

Log contents	Database
<T_0, start>	
<T_0, A, 1000, 950>	A = 950
<T_0, B, 2000, 2050>	B = 2050

Log contains <T_0, start>, but doesn't contain <T_0, commit>. Hence, T_0 must be undone and the values of A and B are restored to 1000 and 2000 respectively.

Case 2

If failure occurs just after log record for write (C)

Log	Database
<T_0 , start>	
<T_0, A, 1000, 950>	A = 950
<T_0, B, 2000, 2050>	B = 2050
<T_0, commit>	
<T_1, start>	
<T_1 , C, 700, 600>	C = 600

Log contains <T_0, start> and <T_0 , commit>, hence, redo T_0. But the log does not contain <T_1, commit>, hence, Undo T_1 and value of T_1 is restored to original value (old value).

Case 3

If system crashes just after the log record
<T_1, commit>

Log	Database
<T_0 ,start>	
<T_0, A, 1000, 950>	A = 950
<T_0, B, 2000, 2050>	B = 2050
<T_0, commit>	
<T_1, start>	
<T_1, C, 700, 600>	C = 600
<T_1, commit>	

Here, both transactions have to be redone as both have their commit point. The values of A, B, and C are 950, 2050 and 600 respectively.

Example

The following log has a system crash with 4 transactions. Specify, which transactions are rolled back, which operations in the log are redone and which (if any) are undone:

<T_1, start>
<T_1, read A>
<T_1, read D>
<T_1, D, 20>
<commit T1 >
<check point>
<T_2, start>
<T_2, read B>
<T_4, B, 12>
<T_4, start>
<T_4, read, D>
<T_4, D, 15>
<T_3, start>
<T_3, C, 30>
<T_4, read A>
<T_4, A, 20>
<T_4, commit>
<T_2, read D>
<T_2, D, 25> ← system crash.

Transactions T_2 and T_3 are to be rolled back as they don't have their commit point. All the write operations in T_2 and T_3 are undone, i.e., $<T_2, B, 12>$, $<T_3, C, 30>$ and $<T_2, D, 25>$. Operations of T_4 have to be redone.

7. Buffer Management

Apr. 2018 – 3M
Write a short note on buffer management.

DBMS application programs require input/output (I/O) operations, which are performed by a component of operating system. These operations normally use main memory *buffers* to match the speed of the processor with the slower secondary storages.

Buffers are also used to minimize the number of I/O operations between main and secondary memories wherever possible.

The *buffers* are the reserved blocks of the main memory. The buffer is divided into *pages*. The unit of storage and access of the database is called as page.

The component of the operating system that performs the assignment and management of these buffer pages is called *buffer manager*. The buffer manager is responsible for the efficient management of the database buffers that are used to transfer (flushing) database related pages between buffer and secondary storage. This is done to enhance access performance.

The buffer manager takes care of reading of pages from the disk(secondary storage) into the buffers(main memory) until the buffers become full and then using a replacement strategy to decide which buffer(s) to write to disk to make space for new pages that need to read from disk. Some of the replacement strategies used by the buffer manager are

i. First-In-first-out(FIFO) and

ii. Least Recently Used(LRU).

8. Relationship between Recovery Management and Buffer Management

1

Oct. 2018 – 3M
What is relationship between Recovery manager and Buffer management?

In case of any type of failures, a transaction must either be aborted or committed to maintain data integrity. Transaction lag plays an important role for database recovery bringing the database into consistent state. In case of failure, transaction represents the basic unit of recovery in a database system.

The *recovery manager* guarantees the atomicity and durability properties of transactions. During recovery from failure, the recovery manager ensures that either all the effects of given transaction are permanently recorded in the database or none of them are recorded.

A transaction begins with successful execution of a <T, BEGIN>(begin transaction) statement. It ends with a successful execution of a COMMIT statement.

The buffer management effectively provides a temporary copy of a database page. Therefore it is used in a database recovery system in which the modifications are done in this temporary copy and the original page remains unchanged in the secondary storage. Both, the transaction log and the data pages are written to the buffer pages in main memory.

The COMMIT transaction operation takes in two phases, and thus it is called as two-phase commit.

In the first phase of COMMIT operation, the transaction log buffers (the log tail) is written onto the secondary storage.

In the second phase of COMMIT operation, the transaction data buffers (database pages updated by the current transaction) are written to the secondary storage.

8.1 Write-Ahead Logging Protocol (WAL)

WAL is a logging protocol used by the recovery manager to ensure the atomicity and durability properties of a transaction.

The protocol states that all successful updates/writes of a transaction, must be made permanent, before they are applied to the corresponding pages in secondary storage.

This implies that whenever a transaction issues a write operation, the following sequence must be followed:

i. Write an update log record to the log tail, and write the main memory pages corresponding to the log tail to the secondary storage(force write).

ii. Make the corresponding update to the database page in either main memory buffer or the secondary storage.

This can be illustrated by the following example

Imagine a program that is in the middle of performing some operation when the machine it is running on loses power. Upon restart, that program might well need to know whether the operation it was performing succeeded, half-succeeded, or failed.

If a write-ahead log is used, the program can check this log and compare what it was supposed to be doing when it unexpectedly lost power to what was actually done.

On the basis of this comparison, the program could decide to undo what it had started, complete what it had started, or keep things as they are.

In case of data buffer is being used by another transaction, the writing of the page is delayed. Thus, it does not cause any problem because the log is always forced during the first phase of the COMMIT. Since no uncommitted modifications are reflected in the database, the undoing transaction log is not required in this method of database recovery.

8.2 Architecture of the Relationship between RM and BM

Here we use architectural model and discuss the specific interface between the Recovery Manager (RM) and the Buffer Manager (BM). RM is implemented as a component of the DBMS and all accesses to the database are via the Buffer Manager, in coordination with the Recovery manager.

Now, we assume that the database is stored permanently on secondary storage, which in this context is called the *stable storage*. The stability of this storage medium is due to its robustness to failures.

Secondary Storage

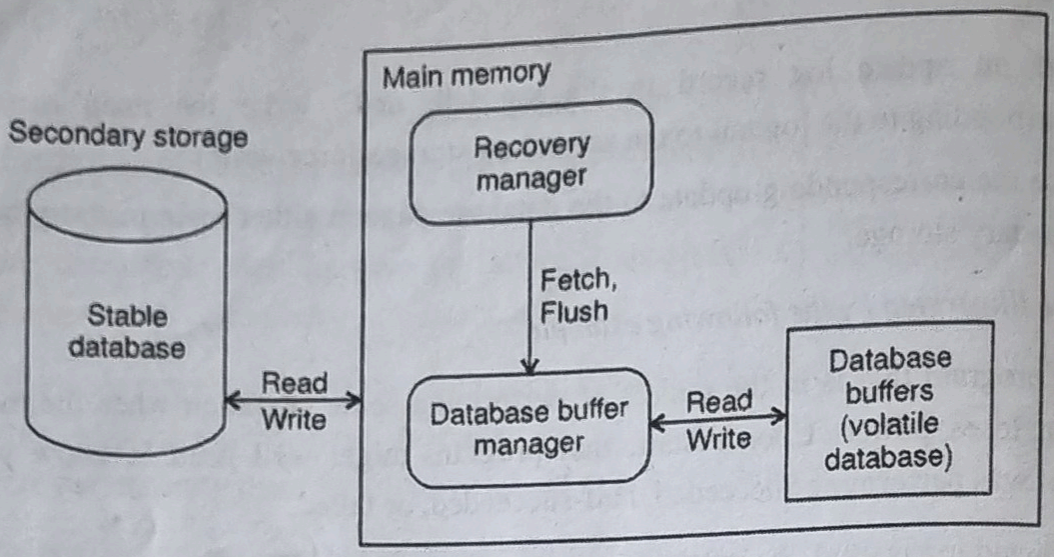


Figure 4.2: Interface between the Local Recovery Manager and the Buffer Manager

A stable storage device would experience considerably less-frequent failures than a non-stable storage device. The version of the database that is kept on stable storage is called as *stable database*. The unit of storage and access of the stable database is typically a *page*.

The buffer manager keeps some of the recently accessed data in main memory buffers. This is done to enhance access performance. Typically, the buffer is divided into *pages* that are of the same size as the stable database pages. The part of the database that is in the buffer is called the *volatile database*. It is important to note that the RM executes the operations on behalf of a transaction only on the volatile database, which, at a later time, is written back to the stable database.

When the RM wants to read a page of data on behalf of a transaction it issues a fetch command, indicating the page that it wants to read. The buffer manager checks to see if that page is already in the buffer (due to a previous fetch command from another transaction) and if so, makes it available for that transaction; if not, it reads the page from the stable database into an empty database buffer. If no empty buffers exist, it selects one of the buffer pages (based on a buffer page replacement algorithm) to write back to stable storage and reads the requested stable database page into that buffer.

The buffer manager also provides the interface by which the RM can actually force it to write back some of the buffer pages. This can be accomplished by means of the flush command, which specifies the buffer pages that the RM wants to be written back.

The buffer manager acts as a medium for all access to the database via the buffers that it manages. It provides this function by fulfilling three tasks:

i. Searching the buffer pool for a given page;

ii. If it is not found in the buffer, allocating a free buffer page and loading the buffer page with a data page that is brought in from secondary storage;

iii. If no free buffer pages are available, choosing a buffer page for replacement.

There are 5 commands that form the interface of Buffer manager to RM

Begin_transaction, read, write, commit and abort commands. The sixth interface command to the RM which is used during recovery from system failures, is: recover.

The execution of some of these commands (specifically, abort, commit, and recover) is quite dependent on the specific RM algorithms that are used as well as on the interaction of the RM with the buffer manager. Others (i.e., begin transaction, read, and write) are quite independent.

8.3 The Steal and the Force Approach

There are two decisions involved in the implementation of the local recovery manager, the buffer manager, and the interaction between the two components is whether or not the buffer manager obeys the recovery manager's instructions as to when to write the database buffer pages to stable storage.

i. The first one is whether the buffer manager may write the buffer pages updated by an uncommitted transaction onto stable storage, or should it wait for the RM to instruct it to write them onto the stable storage. We call this the *fix/no-fix* decision. The reasons for the choice of this terminology will become apparent shortly. Note that it is also called the **steal/no-steal** decision.

ii. The second decision is whether the RM will force the buffer manager to flush the buffer pages updated by a transaction onto the stable storage, when the transaction commits (i.e., the commit point), or is the buffer manager free to flush them out onto the stable storage whenever it needs to, according to its buffer management algorithm. We call this the *flush/no-flush* decision. It is called the *force/no-force* decision.

Accordingly, four alternatives can be identified

i. no-fix/no-flush,(steal/no-force)

ii. no-fix/flush,(steal/force)

iii. fix/no-flush,(no-steal/no-force) and

iv. fix/flush(no-steal/force).

9. Database Backup and Recovery from Catastrophic Failures

1

Apr. 2018 – *1M*
What do you mean by backup?

Recovery from catastrophic failure is based on restoring a database backup copy by redoing operations of committed transactions (stored in archived log file) upto the time of failure.

Backup and recovery functions constitute a very important component of DBMS.

There are functions within the DBMS to schedule automatic database backups to permanent secondary storage device, like tapes or disks.

There are different levels of backups like

i. A full backup of the database or dump of the database.

ii. A differential backup of the database in which only the last modifications done to the database are copied.

iii. A backup of the transaction log only; this level backs up all the transaction log operations that are not reflected in the previous backup copy of the database.

The database backup is always stored at a secured place and protected against dangers such as theft, flood, fire, etc.

10. Shadow Paging

Oct. 2017 – 4M
Write a short note on shadow paging.

Shadow paging considers the database to be composed of a number of fixed size disk pages or disk blocks for recovery purpose.

Shadow paging technique maintains two directories (two for each database page) during life of a transaction a current directory and a shadow directory. When the transaction begins, the two directories are identical. The shadow directory is saved on the disk and the current directory is used by the transaction. The shadow directory is never altered and is used to restore the database in the case of a failure as shown in *fig.4.3(a)*.

Shadow directory is never modified during the transaction execution. When a write operation is performed on a data item, a new copy of the modified database page is created but the old copy of that page is not overwritten. The new copy is written somewhere else.

The current directory entry is modified to point to the new disk block. The shadow directory keeps pointing to the old unmodified disk block as it is not updated and the old disk block is not over written.

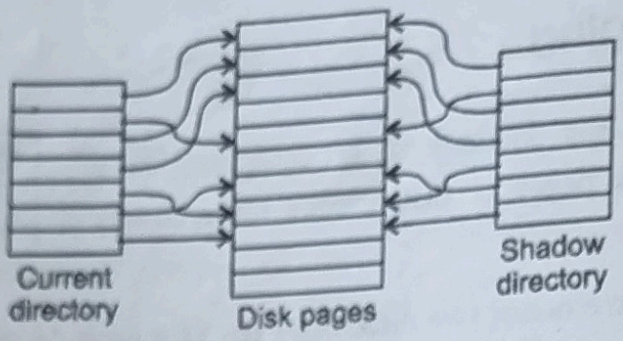
The concept of shadow paging is shown in *fig. 4.3(b)*.

For pages, changed by the transaction, the two versions of the database page are kept one referenced by the current directory, and the other by the shadow directory.

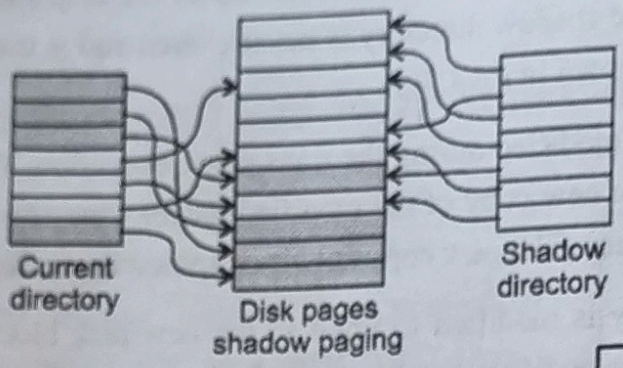
The updated pages are shown by shaded portion in *fig.4.3(b)*.

To recover from a transaction failure, it is sufficient to free the modified database page and discard the current directory. The state of the database before the start of the transaction is available through the shadow directory and that state is recovered by reinstalling the shadow directory.

Committing a transaction is done by discarding the shadow directory. Thus the current directory is used to record all updates to the database. When the transaction is completed, the current directory becomes the shadow directory since the recovery using shadow paging technique required neither an undo nor a redo, it is classified as NO – UNDO/NO – REDO algorithm.



a. **Shadow and current directories and disk pages before transaction execution**



b. **Shadow paging**

Figure 4.3

1

Oct. 2018 – 4M
What is shadow paging? State any two disadvantages of shadow paging.

Advantages of Shadow-page Technique

i. Shadow paging is an alternative to log based crash-recovery (techniques).

ii. This scheme is useful if transactions execute serially.

iii. Recovery from crashes is significantly faster using shadow paging.

Drawbacks to the Shadow-page Technique

i. **Commit overhead:** The commit of a single transaction using shadow paging requires multiple blocks to be output -- the current page table, the actual data and the disk address of the current page table. Log-based schemes need to output only the log records.

ii. **Data fragmentation:** Shadow paging causes database pages to change locations therefore, no longer contiguous.

iii. **Garbage collection:** Each time that a transaction commits, the database pages containing the old version of data changed by the transactions must become inaccessible. Such pages

are considered to be *garbage*. Since they are not part of the free space and do not contain any usable information.

Periodically it is necessary to find all of the garbage pages and add them to the list of free pages. This process is called *garbage collection* and imposes additional overhead and complexity on the system.

Solved Examples

1. Consider the log given below corresponding to a schedule S. Specify which transactions are rolled back, which operations in the log are redone and which (if any) are undone.

<Start–transaction, T_1>
<read–item, T_1, A>
<read–item, T_1, D>
<write–item, T_1,D, 20>
<commit, T_1>
<check point>
<Start–transaction, T_2>
<read–item, T_2,B>
<write–item, T_2, B, 12>
<Start–transaction, T_4>
<read–item, T_4, B>
<write–item, T_4, B, 15>
<Start–transaction, T_3>
<write–item, T_3, A, 30>
<read–item, T_4, A>
<write–item, T_4, A, 20>
<commit, T_4>
<read–item, T_2, D>
<write–item, T_2, D, 25> ← System Crash.