RECOVERY AND ATOMICITY

INTRODUCTION

The process of restoring the database to a correct state in the event of a failure is known as database recovery. It is a service to be provided by DBMS to ensure the database ***reliability***. Reliability here means both, the resilience of DBMS to various types of failure and its capability to recover from them. T**he database has to be restored back to consistent state from its inconsistent state that has been caused due to failure.**

CATEGORIZATION OF RECOVERY ALGORITHMS

Conceptually, there are two main techniques for recovery from non-catastrophic transaction failures:

1. Deferred update (or NO UNDO/REDO) algorithm.
2. Immediate update (or UNDO/REDO) algorithm

Deferred update : These techniques do not physically update on disk until after a transaction reaches its commit point. Then the updates are recorded in the database. Before reaching commit, all transaction updates are recorded in the local transaction work space or buffers. During commit, the updates are first recorded persistently in the log and then written to the database.

If a transaction fails before reaching its commit point, it will not have changed the database in any way, so UNDO is not needed. It may be necessary to REDO the effect of the operations of a committed transaction from the log, because their effect yet have been recorded in the database. **Hence, deferred update is also known as the NO-UNDO/REDO algorithm.**

Immediate update : The database may be updated by some operations of a transaction before the transaction reaches its commit point. However, these operations are typically recorded in the log on disk by force writing before they are applied to the database , making recovery still possible**. If a transaction fails after recording some changes in the database but before reaching its commit point, the effect of its operations on the database must be undone** i.e. the transaction must be rolled back. H**ere both undo and redo may be required during recovery. This technique is known as UNDO/REDO algorithm.**

TERMINOLOGIES USED

The recovery processes are very much similar to our O.S function like that of buffering and caching of disk pages in main memory. In general, it is convenient to consider recovery in terms of the database disk pages (or blocks). Disk is partitioned into fixed length storage units called as blocks. **Please note that blocks are the units of data transfer to and from disk and may contain several data.** When a transaction starts, the data required for the transaction is transferred from the disk to the main memory in blocks. When all operation from main memory to disk for final storage of results. **The blocks residing on the disk are called as physical blocks. The block residing temporarily on main memory are called as buffer blocks. The area of memory where block resides temporarily is called as disk buffer.**

Associated with each buffer in the cache is a **dirty bit.** A directory for the cache is used to keep track of which database items are in the buffers. This **dirty bit** can be included in the directory to show whether or not the buffer has been modified. Another bit called as **pin-unpin** bit is also needed. A page in cache is pinned(i.e., bit-value =1) if it cannot be written back to disk as yet.

There are 2 methods to flush a modified buffer back to disk. They are as follows :

1. In-place updating
2. Shadowing

1. **In- place updating** will write the buffer back to the same original disk location. So, it over-writes the old value of any changed items on disk. Hence, a single copy of each database disk block is maintained.
2. **Shadowing :** It writes an updated buffer at a different disk location, so multiple versions of data items can be maintained. **In general, the old value of the data item before updating is called as the before-image (BFIM) and the new value after updating is called as the after-image (AFIM). Here in this method, both AFIM and BFIM can be kept on disk. So,** it is not necessary to maintain a log for recovering.

The following terminology is used in database recovery when pages are written back to disk:

(a) **Steal policy :** It allows the buffer manager to write a buffer to disk before a transaction commits. That is, the buffer is **unpinned** (page can be written back to disk). In other words, the buffer manager **"steals"** a page from the transaction. The alternative policy is **no-steal.**

(b) Force policy : It ensured that all pages updated by a transaction are immediately written to disk when the transaction commits. The alternative policy is **no force.**

Which approach to use ?

We use a **no-steal, force policy**. With **no-steal**, we do not have to **undo changes** of an aborted transaction because the changes will not have been written to disk and with force. We **do not** have to **redo the changes** of a committed transaction if there is a subsequent crash because all the changes will have been written to disk at commit.

For example, the deferred update recovery protocol uses no-steal policy.

On the other hand, the **steal policy** avoids the need for a very large buffer space to store all updated pages by a set of concurrent transactions which in practice may be unrealistic. Also the **no-force** policy has the distinct advantage of not having to rewrite a page to disk for a later transaction that has been updated by an earlier committed transaction and may still be in a database buffer. ***This is the reason as the why most of the DBMS employ a steal, no-force policy***.

To achieve our goal of atomicity, we firstly need to output the information regarding modifications to a stable storage without modifying the database. There are 2 ways to perform such outputs :

(a) Log-based recovery.
(b) Shadow paging.

LOG-BASED RECOVERY

It is the most common structure used for recording any modifications made to the database. In this recovery procedure, a **log-file** is maintained. A log file is a sequence of four types of log record, namely

(i) <Start > Log Record
(ii) <Update> Log Record
(iii) <Commit> Log Record
(iv) <Abort> Log Record

(i) **<Start> Log Record :** It contains information about the start of each transaction. It has a transaction identifier, which is always unique. It's **syntax i**s

$<T_i$**,** Start>

(ii) **<Update> Log Record :** It describes a single database write. It's **syntax** is

$<T_i, X_j, V1, V2>$

Where T_i - transaction identifier
X_j - data item
V1 - old data-item value
V2 - New or modified value of the data item.

(iii) **<Commit> Log Record:** When a transaction, T_i, is successfully committed or completed, a $<T_i$, commit> log record is stored in the log file.

(iv) **<Abort> Log Record :** When a transaction. T_i is aborted due to any reason, a $<T_i$, abort> log record is stored in the log file.

whenever a transaction performs a write operation, it is necessary that the log record for that write be created before the database is modified. Once a log record exists, we have **the ability to undo a transaction.** So a log record plays a vital role during recovery process of the database. Hence, it must reside in stable storage. As soon as a log record is created, it must be written to stable storage. There are two technique for log-based recovery:

1. **Deferred Database Modification**
2. **Immediate Database Modification**

1. **Deferred Database Modifications: it ensures transaction atomicity by recording all database modification in the log, by deferring the execution of all write operations of a transaction until the transaction partially commits**.

A transaction is said to be partially committed once the final action of the transaction has been executed. When a transaction has performed all the actions, then the information in the log associated with the transaction is used in execution of the deferred writes. In other words, at partial commits time logged update are **"replayed**" into database items.

It means that in this case during write operation the modified values of local variable are not copied into database items, but the corresponding new and old values are stored in the log record. But when the transaction successfully performed all the operations then the information stored at log record is used to set the value of data items, but if the transaction fail to complete, then the value of data item retain their old values and hence in that case information on the log record is simply ignored.

The execution of transaction T_i proceeds as follows :

<T_i, Start> Before T_i starts its execution, a log record is written to the log file

<T_i, A, V2> The write operation by T_i results in the writing of new records to the log. This records indicates the new value of A i.e, V2 after the write operation performed by T_i

<T_i Commit> **When T_i partially commits this record is written to the log.**

It is important to note that only the new value of the data item is required by the deferred modification technique because if the transaction fails before the commit then log record is simply ignored because data item already contain old value due to deferred write operation. But if the transaction commit, the new value present in the log record are copied to the data base item. Thus log record contain only the new values of data item, not the old values of data item. This approach save the stable storage space and reduce the complexity of log record.

Example 1. Consider two transactions, T_1 and T_2 as follows:

T_1	T_2

Read (A, a);
a = a-100;
write (A, a);
Read (B, b);
b = b + 100;
write(B, b)
	Read(C, c);
	c = c -200;
	Write (C, c);

Suppose T_1 and T_2 are executed serially i.e., $T_1 \longrightarrow T_2$ Assume the values of A, B and C before the execution as 1000, 2000 and 3000 respectively. What will be the state of Log Record and Database? What happens if a crash occurs:

(a) just after write (B b)
(b) just after write (C, c)
(c) just after <T_2, Commit>.
(d) During recovery action.

Solution. The result of the execution of T_1 and T_2 is as follows :

Log	Database	Sequence of Time
<T_1, start> (T_1, A,900> (T_1, B,2100> (T_1, Commit> <T_2, Start> <T_2, C,2800> <T_2, Commit	(Buffer) A = 900 B = 2100 We do not known Whether A B and are there on disk or not Because the system decides output Operation.	↓

Here the value of A is changed in the database only after the record <T1, A, 900> has been placed in the log. By using the log, the system can handle any failure that results in the loss of information on volatile storage.

*The **recovery procedure*** of deferred database modification is based on the **Redo operation,** Redo (T_i). It sets the value of all data items updated by T_i to the new values from the log of records. After a failure has occurred the recovery subsystem consults the log to determine which transaction need to be redone. **Here transaction, Ti, needs to be redone if the log contains both the record <Ti, start> and the record <Ti, Commit>. Also if the system crashes after the transaction completes its execution, then the information in the log is used in restoring the system to a previous consistent state.** So use redo operation to get the modified values from the log record. Otherwise, ignore the log record and re-execute the transaction.

Now, say, system crashes before the completion of the transaction :

(a) After write (B,b) : If this is the case, then the log record is as follows :

<T_1, Start>
<T_1, A, 900>
<T_1, B, 2100.

Since no commit record appears in the log, it means that no write operation is performed on the database item. So, database items just retain their old values. So, A and B are 1000 and 2000 respectively. Thus, when the system comes back, no redo actions need to be taken. the log records of the incomplete transaction, T_1, can be deleted from the log

(b) After write (C, c) : If this is the case, then the log record state is as follows :

<T_1, Start>
<T_1, A, 900>
<T_1,B,2100>
<T_1, Commit>
<T_2, Start>
<T_2,C,2800>

Since T_1 is completed successfully before the crash, when the system comes back then the new value of items must be copied from the log record to the data items. It means that redo (T_1) must be performed to recover the system after crash. The T_2 transaction is not committed, so there is no need of redo (T_2) operation. T_2 is re-executed. Again, the log record of the incomplete transaction,(T_2), can be deleted for the log.

So, redo (T_1) is done as the record <T_1, Commit > appears in the log i.e., A = 900 and B =2100. T_2 must be re-executed and C = 3000.

(c) **Just after <T_2, Commit> :** In this case, the state, of the log record is as follows :

<T_1, Start>
<T_1, A, 900>
<T_1, B, 2100>
<T_1, Commit>
<T_2, Start>
<T_2, C,2800>
<T_2, Commit>

Here when the system come back, there are 2 commit record in the log-one from T_1 and another from T_2, so, the operation redo (T_1) and redo (T_2) must be performed. So now values of A,B and C are 900,2100 and 2800 respectively.

(d) **During recovery action :** In this case, some transactions recover and some may not. To remove such problems, recovery actions are restarted from the beginning.

2. Immediate Database Modification: The immediate database modification technique allows database modification to be output to the database while the transaction is still in the active state. The data modification written by active transactions are called "uncommitted modification".

If the system crash or transaction aborts, then the old value field of the log records is used to restore the modified data item to the value they had prior to the start of the transaction. This restoration is accomplished through the **undo operation**. In order to understand undo operation, let us consider the format of log record.

<T_i, X_j, V_{old}, V_{new}>

Here, Ti is transaction identifier, X_j is the data item, V_{old} is the value of data item and V_{new} is the modified or new value of data item X_j.

Undo (T_i) : It restores the value of all data items updated by transaction T1 to the old values. Before a transaction T1 starts its execution the record <T1, start> is written to the log. During its execution, any write (x) operation by T1 is performed by writing of the appropriate new update record to the log. When T1 partially commits the record <Ti, commit> is written to the log.

Example 2. *Consider the two transactions, T_1 and T_2 as shown below :*

T_1	T_2
Read (A,a); a = a – 100; Write (A, a); Read (B, b); b = b + 100 Write (B,b);	Read (C,c) c = c – 200; Write (C, c);

If we use immediate database modification technique then what will be the contents of log and Database.

What happens if crash occurs :

(a) just after write (B, b)

(b) Just after write (C, c)

(c) Just after <T2, commit>?

Solution. The contents of Log and Database are as follows :

Log	Database	Sequence of Time
<T_1, start> (T_1, A, 1000, 900> (T_1, B, 2000, 2100> (T_1, Commit> <T_2, Start> <T_2, C, 3000, 2800> <T_2, Commit>	(Buffer) A = 900 } Uncommited update. If T is aborted, new Values must be replaced by old Values. B = 2100 C = 2800	↓

Using the log the system can handle any failure that result in the loss of information of volatile storage. *The recovery scheme uses two-procedures :*

Undo (T_i) : Restores the value of all data items updated by transaction T_i to the old values. The set of data items updated by T_i and their respective old values can be found in the log.

After a failure has occurred, the recovery scheme consults the log record to determine which transaction needs to be undone. Transaction Ti needs to be undone if the log contains the record <T_i, start> but does not contain the record <T_i, commit>. This transaction is crashed during execution. Thus. Transaction T_i needs to be undone.

Redo (T_i) : Sets the values of all data items updated by transaction T_i to the new values. These new values can be found in log record. After a failure has occurred log record is consulted to determine which transaction need to be redone.

Transaction T_i needs to be redone if the log contain both the record <T_i start> and the record <T_i, commit>. This transaction is crashed just after partially committed. Thus transaction Ti needs to be redone.

When a **system crashes** before the completion of transaction:

(a) **Just after write (B,b) :** When the system comes back, it finds the record <T_1, start> in the log, but no corresponding <T_1, commit> record. Thus, T_1 must be undone. It means that the old values are copied from the log record to database items. So T_1 is undone. The old values of A and B are taken from the log record T_1 and T_2 must be re-executed.

(b) **Just after write (C, c)** : Here, when the system comes back, two recovery actions need to be taken :

(i) Undo (T_2) must be done since the record <T_2, start> appears in the log but there is no record <T_2, commit>.

(ii) Redo (T_1) must be done as the log contains both the records <T_1, start> and record <T_1, commit>. So, now a = 900, B = 2100 and C = 3000. T_2 must be re-executed.

(c) **After <T_2, commit> :** When the system comes back, transactions, T_1 and T_2 need to be redone because the record <T_1, start> and T_1, commit> appears in the log as do the record <T_2, start> and <T_2, commit>. So A =900, B =2100 and C = 2800

SHADOW PAGING

Lorie in 1977 introduced this technique of shadow paging as an alternative to log-based recovery schemes. ***The database is partitioned into some number of fixed-length blocks called as pages or disk blocks.*** The pages are mapped into physical blocks of storage by means of a page table, with one entry for each logical page of the database. Suppose, there are n pages. There must be a way to find the i^{th} page of the database for any given i. We use a page table for this purpose. The page-table has n entries one for each database page. Please note that each entry has a pointer to a page on the disk, as shown in Fig. below.

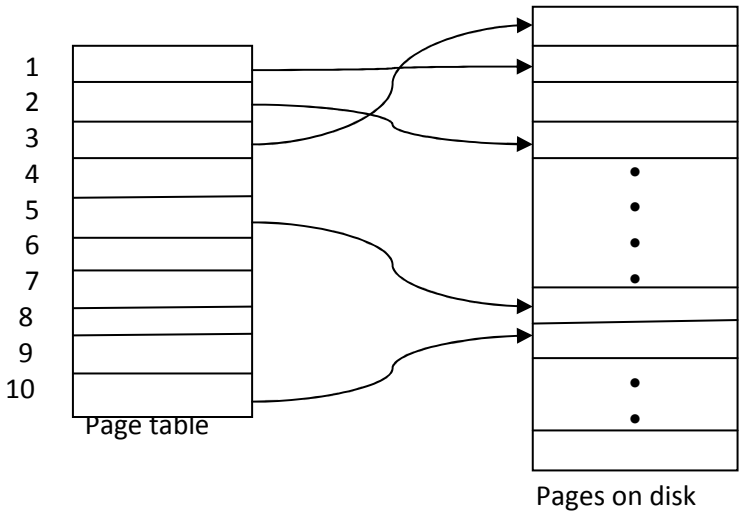


Fig. Page table and disk pages

So, the first entry contains a pointer to the first page of the database, the second entry to the second page and so on.

Working : the key idea here is to maintain two pages tables during the life of a transaction :

(a) Current page table.
(b) Shadow page table.

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

So, we must **remember that** out of these two tables **– shadow table / directory is saved to the disk and the current table/directory is used by the transaction, T. Here shadow directory is never changed and is basically used to restore the database in case of a failure**. Shadow paging is a technique for providing atomicity and durability in database systems. The page size is of the order 2^{10} to 2^{15} bytes. When a transaction completes, the current page table becomes the shadow page table.

How the write operation is performed?

Suppose, a transaction, T, performs a write (X) operation and that X resides on the i^{th} page. The write operation is executed as follows :

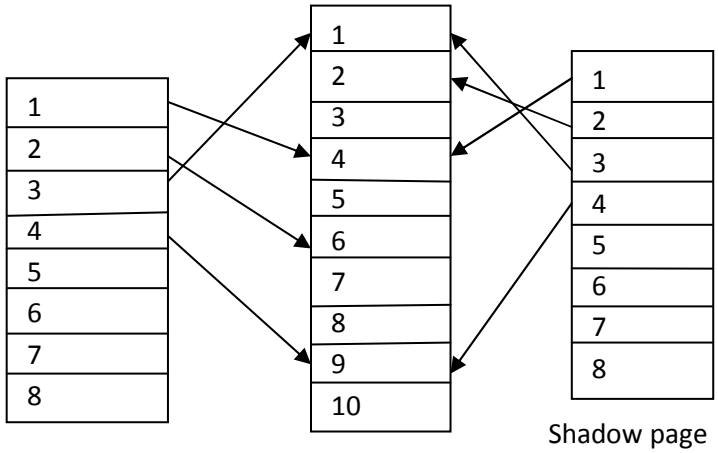
Step 1. If the i^{th} page (i.e., page where X resides) is not already in main memory, then issue input (I).

Step 2. If first write is being done on the i^{th} page by transaction, T, then modify the current page table as follows:

(a) Find an unused page on disk DBMS has access to a list of unused (free) pages.
(b) Delete the page found in step 2(a) above from the list of free page frames.
(c) Modify the current page table such that the i^{th} entry points to the page found in step 2(a) above.

Step 3. Assign the value of X_j to X in the buffer page.

If we compare this technique with **log-based recovery** we find that in log-based method the system writes the data item in the same location (overwriting). So, only a single copy of each item is maintained on disk. On the other hand, in **shadowing,** the system writes a new data item at a different disk location thereby creating many copies of the data item. This is shown in Fig



Write operation on page -2

We Store the shadow page table on non-volatile storage so that the state of the database prior to the execution of the transaction can be recovered in the event of a crash or transaction abort. When a transaction commit, the current page table is written to a non-volatile storage, as it provides the only means of locating database pages. **The current page table may be kept in the main memory i.e., volatile storage,** So even if the current page table is lost during crash, the system recovers using shadow page table.

Use of shadow table when system crashes : One simple method is to choose one fixed location in stable storage that contains the disk address of the shadow page table. After a crash, we copy the shadow page table into the main memory and then use it for subsequent transaction processing. T**he shadow page table will point to be database pages corresponding to the state of the database prior to any transaction that was active at the time of the crash. This is the reason as to why aborts are automatic.** There is no need of invoking undo operations as required in log-based technique.

How commit operation is preformed?

1. Flush all modified pages in main memory to disk.
2. Output current page table to disk.
3. 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 i.e., new transaction 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 Technique : Shadowing has some advantages and are given below :
   1. The overhead of maintaining the transaction log file is eliminated.
   2. Since there is no need for undo or redo operations, recovery from crashes is significantly faster.
   3. The recovery process is simpler.

   Disadvantages of Shadow Paging : Shadowing has some disadvantages also and are discussed below :
   1. **Scattering (or Data Fragmentation)** : In this shadowing technique, it is difficult to keep related database pages closer, together on the disk. This results in **scattering of data.**
   2. **Garbage collection :** Each time that a transaction commits, the database pages containing the old version of data changed by the transaction become inaccessible. Such pages are considered as **garbage** because they are not part of the free space and do not contain usable information. Garbage is also created during system crashing. So we must periodically find all such garbage pages and old them to the list of free pages. Doing this, naturally adds to more of overhead and complexity on the system.
   3. **Commit overhead :** During shadowing, the commit of a single transaction involves three steps :
      (a) Copy the actual data block from RAM to hard disk by output operation.
      (b) Copy the current page table from main memory (RAM) to the disk by output operation.
      (c) Change the disk address of the current page table.

      On the other hand. **Log-based schemes** need to output only log records which fit within one block if transactions are small.

4. ***Non-suitability for multi-user, concurrent applications:*** As we know that for the concurrent application, multiple transaction run in parallel. So, now each transaction needs a separate shadow and current page tables. This makes the system very complex. So shadowing is not suitable for a multi-user system.
   Example Consider a transaction, T, as follows :

T	

Read (A, a)
a = a – 50;
Write (A, a)
Read (B,b);
b = b + 50;
Write (B, b);
	//Say A = 1000,
	B = 2000,
	C = 3000

Explain how shadowing will be done here? What happens if the system crashes :

(a) Before commit of T.
(b) After commit of T?

Solution. When the transaction, T, starts the shadow page table stored on the disk is copied to the main memory and becomes the current page table.

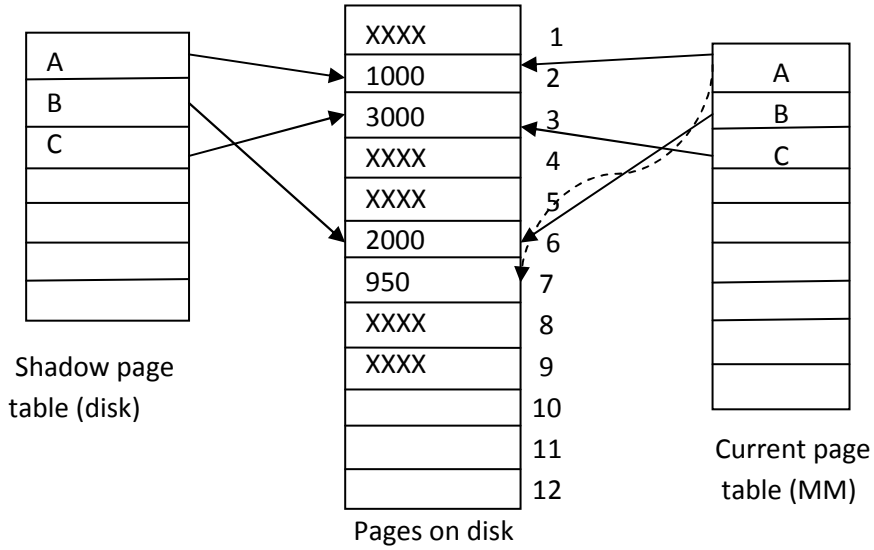
When T is executed, the first instruction is executed as follows :

I. Read (A, a) : Herein, the block containing the data item, a, is shifted from hard disk to RAM by input (X) operation and copies the value A to a i.e., A =1000, so a ⟵1000. But now a = a - 50 (Where a is a local variable). So, a = 1000 – 50 = **950.**

II. Write (A, a) : If the block containing data item, A, is not available in RAM, then input (X) operation is performed.

Find the unused space on the disk e.g., 7th page, shown in Fig. below Delete this page from the list of free page frames. So now the free page list is :

Now, modify the current page-table so that current page table entry for a now points to 7th page instead of page-2



As shown in Fig. above the pointer of A block on current page table now points to 7th page on hard disk (dotted line) instead of 2nd page. So, we assign, 950 value

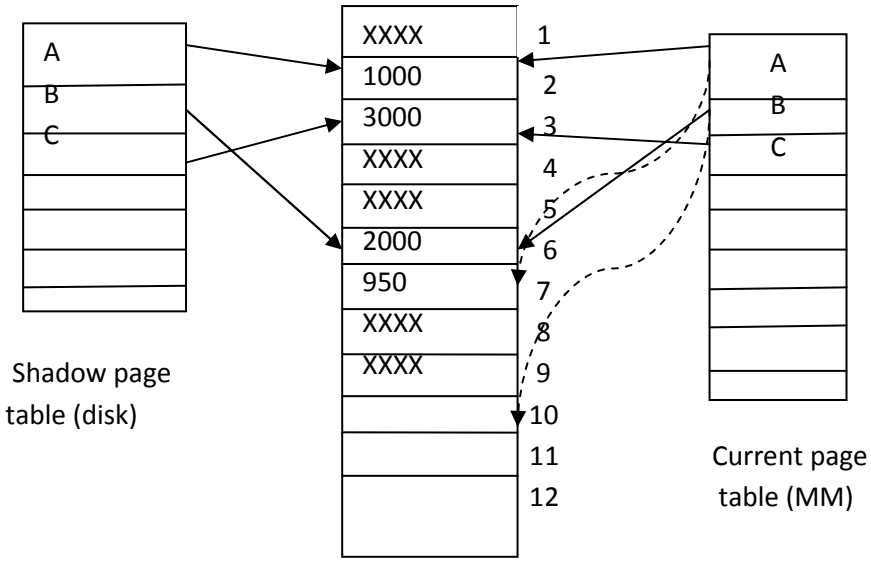
(as a = a-50) ; to data-item A in MM

III. Read (B, b) : The block containing the data item, A, is shifted from hard disk to RAM and the value is copied from B to b.

Now, B = 2000 =b (say, b is a local variable)

b = b + 50 = 2050

IV. Write (B, b) : So, now an unused page is found on the disk, we did before. The next page free available to us is page 10. So, it is selected and the new value of B i.e,. 2050 is copied to page 10. This is shown in Fig. The 10^{th} pages is thus deleted from the list of free pages and the list now becomes (10 ,11, 12). So, we modify the current page table such that B now point to page-10, instead of page-6. This is shown in Fig. 8.4 below.



Please note here, how the B-pointer from the current page table is made to point to page-10, instead of page-6 where initial value of B was 2000. This process goes on.

Finally, **commit** is executed. This setup occurring here are as follows :

1. All pages on RAM that have been changed by T are output to disk i.e., A =950 at page-7 and B = 2050 at page-10.
2. Now, output the current page table to disk without overwriting the shadow page table.
3. Also, output the disk address of the current page table to the fixed location in stable storage containing the address of shadow page table. This changes the address of old shadow page table. So, the current page table, now becomes shadow page table and the transaction is completed.

Now, what happens during system crash? Two cases are given in the question .

(a) **Crash occurs before commit of T :** Now, the current page table stored in the main memory is crashed. When the system comes back. it gets the address of shadow page table from the fixed location on disk. This table points to the old value of A, B and C. **Here this is just equal to the undo operation of the log-based recovery technique.** We can also say that if the system fails before the commit of transaction, T, we will get the previous state of data-item from the shadow table.

(b) **Crash occurs after commit of T :** Now, when the system recovers, it will get the address of shadow page table from the fixed location. This table will now point to the modified value of A and B. This is similar to the **redo operation** of log based recovery technique. We can also say that there is no need of re-executing T and it will recover the data successfully.