8.5.1.6 Lock Conversion

To improve efficiency of two-phase locking protocol, modes of lock can be converted.

(i) *Upgrade* : Coversion of shared mode to exclusive mode is known as upgrade.

(ii) *Downgrade* : Conversion of exclusive mode to shared mode is known as downgrade.

Consider transaction T_5 as shown in Figure 8.12.

In T_5, an exclusive lock of A is needed after sometime so first obtain shared lock on A and then upgrade it. During this period any other transaction can also obtain shared lock of A and hence increase efficiency.

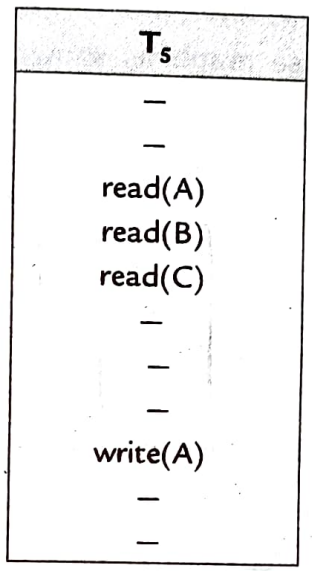

FIGURE 8.12. *Lock conversion (Upgrade).*

8.5.2 Graph Based Protocols

In graph based protocols, an additional information is needed for each transaction to know, how they will access data items of database. These protocols are used where two-phase protocols are not applicable but they required additional information that is not required in two-phase protocols. In graph based protocols partial ordering is used.

Consider, a set of data items D = $\{d_1, d_2, ..., d_i, d_j, ...d_z\}$. If $d_i \rightarrow d_j$ then, if any transaction T want to access both d_i and d_j then T have to access d_i first then d_j.

Here, **tree protocol** is used in which D is viewed as a directed acyclic graph known as *database graph.*

Rules for tree protocol :

1. Any transaction T can lock any data item at first time.
2. After first lock if T wants to lock any other data item v then, first, T have to lock its parent.
3. T cannot release all exclusive locks before commit or abort.
4. Suppose T has obtained lock on data item v and then release it after sometime. Then T cannot obtain any lock on v again.

All the schedules under graph based protocols maintain serializability. Cascadelessness and recoverability are maintained by rule (3) but this reduces concurrency and hence reduce efficiency.

Alternate approach : Here rule (3) is modified. Modified version is (3) T can release any lock at any lock.

But in alternate approach there may be cascading rollback.

For each data item, if any transaction performed the last write to some data item and still uncommitted then record that transaction.

Commit dependency : A transaction T_i has commit dependency upon T_j if T_i wants to read a uncommitted data item v and T_j is the most recent transaction which performed write operation on v.

If any of these transactions aborts, T_i must also be aborted.

Consider the tree structured database graph as shown in Figure 8.13.

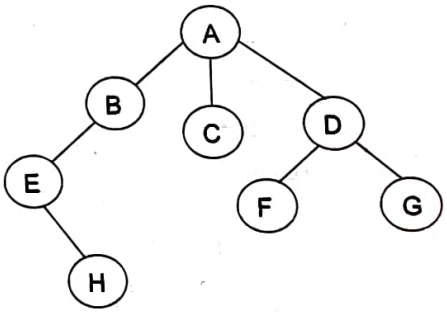

FIGURE 8.13. *Tree structured database graph.*

Suppose we want to operate

$$G = G + 10 \text{ and}$$
$$H = H - 5$$

then possible transaction is as shown in Figure 8.14.

T_6
X-lock(G);
read(G);
G = G + 10;
unlock(G);
S - lock(A);
read(A);
S - lock(B);
read(B);
unlock(A);
S - lock(E);
read(E);
unlock(B);
X - lock(H);
read(H);
H = H - 5;
unlock(E);
unlock(H);

FIGURE 8.14. *Transaction using graph based protocols.*

Advantages

1. There is no deadlock.
2. Unlike in two-phase, data items can be unlocked at any time, that reduces waiting time.
3. More concurrency.

Disadvantages : Those items have to be locked that have no use in transactions. This results in increased locking overhead. (locking of A, B and E to lock H in T_6 of Figure 8.14).

8.5.3 Timestamp-based Protocols

In two-phase and graph based protocols, conflict transactions are found at run time. To order them in advance, use timestamp based protocols.

NOTE *(Conflict transactions are those transactions that need same data items during execution).*

For each transaction T_i, assign a unique fixed number or timestamp before it starts, execution. It is denoted by $TS(T_i)$. For any two transactions T_i and T_j, $TS(T_i) > TS(T_j)$ or $TS(T_j) < TS(T_i)$ but never be equal. Timestamps can be given by using :

(*i*) **System clock :** System clock time is equal to timestamp of transaction.

(*ii*) **Logical counter :** Value of counter is equal to timestamp of transaction. Counter is incremented every time after assigning new timestamp.

Two timestamps are associated with each data item v of database. These are :

(*i*) **Write–TS(v) :** Write timestamp of any data item is equal to the timestamp of most recent transaction that executes **write**(v) successfully and write–TS(v).

(*ii*) **Read–TS(v) :** Read timestamp of any data item is equal to the timestamp of most recent transaction that executed **read**(v) successfully and read–TS(v).

8.5.3.1 The Timestamp-Ordering Protocol

To order conflicting read and write operations use the following rules :

1. **For any transaction T_i, to execute read(v) :**
   (*i*) If $TS(T_i) <$ write–TS(v) then read(v) is rejected and rollback T_i because T_i tries to read the old value of v which is overwritten by any other transaction.
   (*ii*) If $TS(T_i) >$ write–TS(v), then read(v) is executed and read–TS(v) is set to maximum of $TS(T_i)$ and read–TS(v).
2. **For any transaction T_i, to execute write(v) :**
   (*i*) If $TS(T_i) <$ read-TS(v) then T_i tries to write that value of v which was used previously by other transaction. Write(v) is rejected and T_i is rollbacked (to avoid inconsistent analysis problem).
   (*ii*) If $TS(T_i) <$ write-TS(v) then T_i try to write obsolete value of v. So, write(v) is rejected and T_i is rollbacked (to avoid Lost - Update problem).
   (*iii*) Otherwise execute write-TS(v).

Advantages

1. It maintains serializability.
2. It is free from deadlock.
3. It is free from cascading rollbacks.

Disadvantages

1. **Starvation** is possible for long transactions which were forcefully restarted due to conflicting short transactions.

   To avoid starvation, block conflicting transactions for sometime to allow completion of long transactions.

2. Schedules are not recoverable.

8.5.3.2 Thomas Write Rule

To allow more concurrency in timestamp based protocol timestamp ordering protocol is slightly modified and is known as Thomas Write Rule.

In Thomas Write Rule, rules for **read operation** remains unchanged, there is slight variation in **write operation**, that is as follows :

For any transaction T_i, to execute write(v).

1. Same as in timestamp ordering protocol.
2. If $TS(T_i) <$ write-TS(v) then T_i try to write obsolete value of v. So, no need to rollback T_i, just ignored the write operation.

Consider Figure 8.15 which shows the schedule for Transactions T_7 and T_8 and $TS(T_7)<TS(T_8)$.

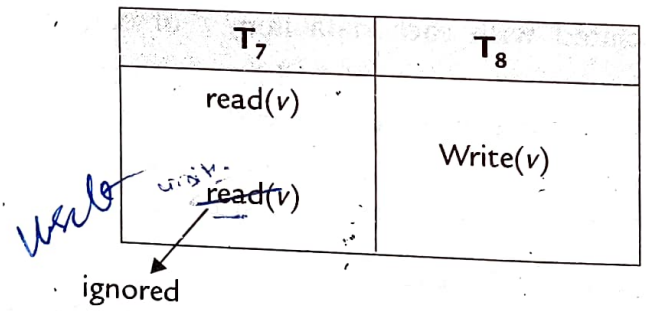

FIGURE 8.15. *Schedule for transactions T_7 and T_8.*

So, T_7 can successfully complete read(v) operation. Then, T_8 can successfully complete write(v) operation. But when T_7 attempts write(v) operation, it is rejected because $TS(T_7) <$ Write-TS(v) **[write-TS(v) = TS(T_8)]**.

But, according to timestamp ordering protocol, T_7 is rolled back which is not required. Since the value of v which T_7 wants to write will never be used.

(*i*) For any transaction T_i such that $TS(T_i)<TS(T_8)$ that want to read the value of v [read (v)] will be rejected and rolled back because $TS(T_i)<$write-TS(v).

(*ii*) For any transaction T_i such that $TS(T_i)>TS(T_8)$ that want to execute read(v) must read the value written by T_8 because read(v) in T_7 is not possible in any way.

So, ignore or delete read(v) in T_7.

8.5.4 Validation Based Protocols

Validation Based Protocols are used where most of the transactions are Read-only and conflicts are low. In these protocols, every transaction T_i have to go through the following phases dependending upon its type :

8.5.5.2 Multiversion Two-phase Locking

In multiversion two-phase locking, advantages of both multiversion concurrency control techniques and two-phase locking technique are combined. It is also helpful to overcome the disadvantages of multiversion timestamp ordering.

Single Timestamp is given to every version of each data item. Here timestamp counter (TS-counter) is used instead of system clock and logical counter. Whenever a transaction commits, TS-counter is incremented by 1.

Read only transactions are based upon multiversion timestamp ordering. These transactions are associated with timestamp equal to the value of TS-counter.

Updations are based upon rigrous two-phase locking protocol.

8.6 DEADLOCKS

A system is said to be in deadlock state if there exist a set of transactions $\{T_0, T_1, ..., T_n\}$ such that each transaction in set is waiting for releasing any resource by any other transaction in that set. In this case, all the transactions are in waiting state.

8.6.1 Necessary Conditions for Deadlock

A system is in deadlock state if it satisfies all of the following conditions.

(i) *Hold and wait :* Whenever a transaction holding at least one resource and waiting for another resources that are currently being held by other processes.

(ii) *Mutual Exclusion :* When any transaction has obtained a nonsharable lock on any data item and any other transaction requests that item.

(iii) *No Preemption :* When a transaction holds any resource even after its completion.

(iv) *Circular Wait :* Let T be the set of waiting processes $T = \{T_0, T_1, ..., T_i\}$ such that T_0 is waiting for a resource that is held by T_1, T_1 is waiting for a resource that is held by T_2, T_i is waiting for a resource that is held by T_0.

8.6.2 Methods for Handling Deadlocks

There are Two methods for handling deadlocks. These are :

8.6.2.1 Deadlock Prevention

It is better to prevent the system from deadlcok condition then to detect it and then recover it. Different approaches for preventing deadlocks are :

1. *Advance Locking :* It is the simplest technique in which each transaction locks all the required data items at the beginning of its execution in atomic manner. It means all items are locked in one step or none is locked.

Disadvantages

1. It is very hard to predict all data items required by transaction at starting.
2. Under utilization is high.
3. Reduces concurrency.

2. *Ordering Data Items :* In this approach, all the data items are assigned in order say 1,2,3,... etc. Any transaction T_i can acquire locks in a particular order, such as in ascending order or descending order.

Disadvantages : It reduces concurrency but less then advance locking.

3. *Ordering Data Items with Two-phase Locking* : In this approach, two-phase locking with ordering data items is used to increase concurrency.

4. *Techniques based on Timestamps* : There are **Two** different techniques to remove deadlock based on time stamps.

(*i*) **Wait-die :** In wait-die technique if any transaction T_i needs any resource that is presently held by T_j then T_i have to wait only if it has timestamp smaller than T_j otherwise T_i is rolled back.

If $TS(T_i) < TS(T_j)$

T_i waits;

else

T_i is rolled back;

It is a nonpreemptive technique.

(*ii*) **Wound-wait :** In wound-wait technique, if any transaction T_i needs any resource that is presently held by T_j then T_i have to wait only if it has timestamp larger then T_j otherwise T_j is rolled back.

If $TS(T_i) > TS(T_j)$

T_i waits;

else

T_j is rolled back;

It is a *preemptive technique.*

Disadvantage : There are unnecessary roll backs in both techniques that leads to starvation.

8.6.2.2 Deadlock Detection and Recovery

In this approach, allow system to enter in deadlock state then detect it and recover it. To employ this approach, system must have :

1. Mechanism to maintain information of current allocation of data items to transactions and the order in which they are allocated.
2. An algorithm which is invoked periodically by system to determine deadlocks in system.
3. An algorithm to recover from deadlock state.

Deadlock Detection : To detect deadlock, maintain **wait-for graph.**

1. *Wait-for Graph* : It consists of a pair G = (V, E), where V is the set of vertices and E is the set of edges. Vertices show transactions and edges show dependency of transactions.

If transaction T_i request a data item which is currently being held by T_j then, an edge $T_i \rightarrow T_j$ is inserted in graph. When T_i has obtained the required item from T_j remove the edge. Deadlock occurs when there is cycle in wait-for graph. Consider wait-for graph as shown in Figure 8.17.

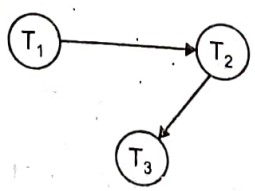

FIGURE 8.17. *Wait-for graph without deadlock situation.*

There are three transactions T_1, T_2 and T_3. The transaction T_1 is waiting for any data item held by T_2 and T_2 is waiting for any data item held by T_3. But there is no deadlock because there is no cycle in wait-for graph.

Consider the graph as shown in Figure 8.18, where T_3 is also waiting for any data item held by T_1.

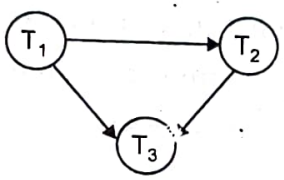

FIGURE 8.18. *Wait-for graph with deadlock.*

Thus, there exists a cycle in wait-for graph.

$$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$$

which results system in deadlock state and T_1, T_2 and T_3 are all blocked.

Now a major question arises : How much will be the time interval after which system invokes deadlock detection algorithm? It depends upon **Two** factors :

1. After how much time a deadlock occurs.
2. How many transactions are deadlocked.

2. *Recovery from Deadlock :* When deadlock detection algorithm detects any deadlock in system, system must take necessary actions to recover from deadlock.

Steps of recovery algorithm are :

1. **Select a victim :** To recover from deadlock, break the cycle in wait-for graph. To break this cycle, rollback any of the deadlocked transaction. The transaction which is rolled back is known as **victim.**

   Various factors to determine victim are :

   (*i*) Cost of transaction.

   (*ii*) How many more data items it required to complete its task.

   (*iii*) How many more transactions affected by its rollback.

   (*iv*) How much the transaction is completed and left.

2. **Rollback :** After selection of victim, it is rollbacked.

   There are **Two** types of rollback to break the cycle :

   (*i*) *Total rollback :* This is simple technique. In total rollback, the victim is rolled back completely.

   (*ii*) *Partial roliback :* In this approach, rollback the transaction upto that point which is necessary to break deadlock. Here transaction is partially rolled back. It is an effective technique but system have to maintain additional information about the state of all running transactions.

3. **Prevent Starvation :** Sometimes, a particular transaction is rolledback several times and never completes which causes starvation of that transaction. To prevent starvation, set the limit of rollbacks or the maximum number, the same transaction chooses as victim. Also, the number of rollback can be added in cost factor to reduce starvation.

Chapter 10

DATABASE RECOVERY SYSTEM

10.1 INTRODUCTION

A Computer system can be failed due to variety of reasons like disk crash, power fluctuation, software error, sabotage and fire or flood at computer site. These failures result in the lost of information. Thus, database must be capable of handling such situations to ensure that the **atomicity** and **durability** properties of transactions are preserved. An integral part of a database system is the *recovery manager* that is responsible for recovering the data. It ensures **atomicity** by undoing the actions of transactions that do not commit and **durability** by making sure that all actions of committed transactions survive system crashes and media failures. The recovery manager deal with a wide variety of database states because it is called during system failures. Furthermore, the database recovery is the process of restoring the database to a consistent state when a failure occurred. In this chapter, we will discuss different failures and database recovery techniques used so that the database system be restored to the most recent consistent state that existed shortly before the time of system failure.

10.2 CLASSIFICATION OF FAILURES

Failure refers to the state when system can no longer continue with its normal execution and that results in loss of information. Different types of failure are as follows :

1. **System crash** : This failure happens due to the bugs in software or by hardware failure etc.
2. **Transaction failure** : This failure happens due to any **logical error** such as overflow of stack, bad input, data not found, less free space available etc., or by **system error** such as deadlocks etc.
3. **Disk failure** : This failure happens due to head crash, tearing of tapes, failure during transfer of data etc.

10.3 RECOVERY CONCEPT

Recovery from failure state refers to the method by which system restore its most recent consistent state just before the time of failure. There are several methods by which you can recover database from failure state. These are defined as follows :

10.3.1 Log Based Recovery

In log based recovery system, a **log** is maintained, in which all the modifications of the database are kept. A log consists of **log records**. For each activity of database, separate log record is made. Log records are maintained in a serial manner in which different activities are happened. There are various log records. A typical update log record must contain following fields:

(i) *Transaction identifier* : A unique number given to each transaction.

(ii) *Data-item identifier* : A unique number given to data item written.

(iii) *Date and time* of updation.

(iv) *Old value* : Value of data item before write.

(v) *New value* : Value of data item after write.

Logs must be written on the non-volatile (stable) storage. In log-based recovery, the following two operations for recovery are required :

(i) *Redo* : It means, the work of the transactions that completed successfully before crash is to be performed again.

(ii) *Undo* : It means, all the work done by the transactions that did not complete due to crash is to be undone.

The redo and undo operations must be idempotent. An idempotent operation is that which gives same result, when executed one or more times.

For any transaction T_i, various log records are :

$[T_i$ start] : It records to log when T_i starts execution.

$[T_i, A_j]$: It records to log when T_i reads data item A_j.

$[T_i, A_j, V_1, V_2]$: It records to log when T_i updates data item A_j, where V_1 refers to old value and V_2 refers to new value of A_j.

$[T_i$ Commit] : It records to log when T_i successfully commits.

$[T_i$ aborts] : It records to log if T_i aborts.

There are *two types* of *Log Based Recovery techniques* and are discussed below :

10.3.1.1 Recovery Based on Deferred Database Modification

In deferred database modification technique, deferred (stops) all the write operations of any Transaction T_i until it partially commits. It means modify real database after T_i partially commits. All the activities are recorded in log. Log records are used to modify actual database. Suppose a transaction T_i wants to write on data item A_j, then a log record $[T_i, A_j, V_1, V_2]$ is saved in log and it is used to modify database. After actual modification T_i enters in committed state. *In this technique, the old value field is not needed.*

Consider the example of Banking system. Suppose you want to transfer Rs.200 from Account A to B in Transaction T_1 and deposit Rs.200 to Account C in T_2. The transactions T_1 and T_2 are shown in Figure 10.1.

T_1	T_2

read(A);
A = A - 200;
write(A);
read(B);
B = B + 200;
write(B);
	read(C);
	C = C + 200;
	write(C);

FIGURE 10.1. *Transactions T_1 and T_2.*

Suppose, the initial values of A, B and C Accounts are Rs. 500, Rs. 1,000 and Rs. 600 respectively, Various log records for T_1 and T_2 are as shown in Figure 10.2.

[T_1 start]
[T_1, A]
[T_1, A, 300]
[T_1 , B]
[T_1, B, 1200]
[T_1 commit]
[T_2 start]
[T_2, C]
[T_2, C, 800]
[T_2 commit]

FIGURE 10.2. *Log records for transactions T_1 and T_2.*

For a **redo operation, log must contain [T_i start] and [T_i** commit] log records.

[T_1 start]
[T_1, A]
[T_1, A, 300]

(a)

[T_1 start]
[T_1, A]
[T_1, A, 300]
[T_1, B]
[T_1, B, 1200]
[T_1 commit]
[T_2 start]
[T_2, C]
[T_2, C, 800]

(b)

FIGURE 10.3. *Log of transactions T_1 and T_2 in case of crash.*

Crash will happen at any time of execution of transactions. Suppose crash happened

(*i*) **After write (A) of T_1:** At that time log records in log are shown in Figure 10.3(*a*). There is no need to redo operation because no commit record appears in the log. Log records of T_1 can be deleted.

(*ii*) **After write (C) of T_2 :** At that time log records in log are shown in Figure 10.3(*b*). In this situation, you have to redo T_1 because both [T_1 start] and [T_1 commit] appears in log. After redo operation, value of A and B are 300 and 1200 respectively. Values remain same because redo is idempotent.

(*iii*) **During recovery :** If system is crashed at the time of recovery, simply starts the recovery again.

10.3.1.2 Recovery Based on Immediate Database Modification

In immediate database modification technique, database is modified by any transaction T_i during its active state. It means, real database is modified just after the write operation but after log record is written to stable storage. This is because log records are used during recovery. Use both Undo and Redo operations in this method. Old value field is also needed (for undo operation). Consider again the banking transaction of Figure 10.1. Corresponding log records after successful completion of T_1 and T_2 are shown in Figure 10.4.

[T_1 start]
[T_1, A]
[T_1, A, 500, 300]
[T_1, B]
[T_1, B, 1000, 1200]
[T_1, commit]
[T_2, start]
[T_2, C]
[T_2, C, 600, 800]
[T_2 commit]

FIGURE 10.4. *Log records for transactions T_1 and T_2.*

- For a transaction T_i to be redone, log must contain both [T_i start] and [T_i commit] records.
- For a transaction T_i to be undone, log must contain only [T_i start] record.

(a)

[T_1 start]
[T_1, A]
[T_1, A, 500, 300]

(b)

[T_1 start]
[T_1, A]
[T_1, A, 500, 300]
[T_1, B]
[T_1 , B, 1000,1200]
[T_1 commit]
[T_2 start]
[T_2, C]
[T_2, C, 600,800]

FIGURE 10.5. *Log of transactions T_1 and T_2 in case of crash.*

Crash will happen at any time of execution of transaction. Suppose crash happened.

(*i*) **After write (A) of T_1 :** At that time log records in log are shown in Figure 10.5(*a*). Here only [T_1 start] exists so undo transaction T_1. As a result, Account A restores its old value 500.

(*ii*) **After write (C) of T_2 :** At that time log records in log are shown in Figure 10.5(*b*). During back-up record [T_2 start] appears but there is no [T_2 commit], so undo transaction T_2. As a result, Account C restores its old value 600. When you found both [T_1 start] and [T_1 commit] records in log, redo transaction T_1 and account A and B both keep their new value.

(*iii*) **During recovery :** If system is crashed at the time of recovery simply starts recovery again.

10.3.1.3 Checkpoints

Both the techniques discussed earlier ensures recovery from failure state, but they have some **disadvantages** such as :

(*i*) They are time consuming because successfully completed transactions have to be redone.

(*ii*) Searching procedure is also time consuming because the whole log has to be searched.

So, use checkpoints to reduce overhead. Any of previous recovery techniques can be used with checkpoints. All the transactions completed successfully or having [T_i commit] record before [checkpoint] record need not to be redone.

During the time of failure, search the most recent checkpoint. All the transactions completed successfully after checkpoint need to be redone. After searching checkpoint, search the most recent transaction T_i that started execution before that checkpoint but not completed. After searching that transaction, redo/undo transaction as required in applied method.

Advantages

1. No need to redo successfully completed transactions before most recent checkpoints.
2. Less searching required.
3. Old records can be deleted.

10.4 SHADOW PAGING

Shadow Paging is an alternative technique for recovery to overcome the disadvantages of log-based recovery techniques. The main idea behind the shadow paging is to keep two page tables in database, one is used for *current operations* and other is used *in case of recovery*.

Database is partitioned into fixed length blocks called *pages*. For memory management, adopt any paging technique.

Page Table

To keep record of each page in database, maintain a page table. Total number of entries in page table is equal to the number of pages in database. Each entry in page table contains a pointer to the physical location of pages.

The two page tables in Shadow Paging are :

(i) *Shadow page table* : This table cannot be changed during any transaction.

(ii) *Current page table* : This table may be changed during transaction.

Both page tables are identical at the start of transaction. Suppose a transaction T_i performs a write operation on data item V, that resides in page *j*. The write operation procedure is as follows :

1. If j^{th} page is in main memory then OK otherwise first transfer it from secondary memory to main memory by instruction input (V).
2. Suppose page is used first time then :
   (*a*) System first finds a free page on disk and delete its entry from free page list.
   (*b*) Then modify the current page table so that it points to the page found in step 2(*a*).
3. Modify the contents of page or assign new value to V.

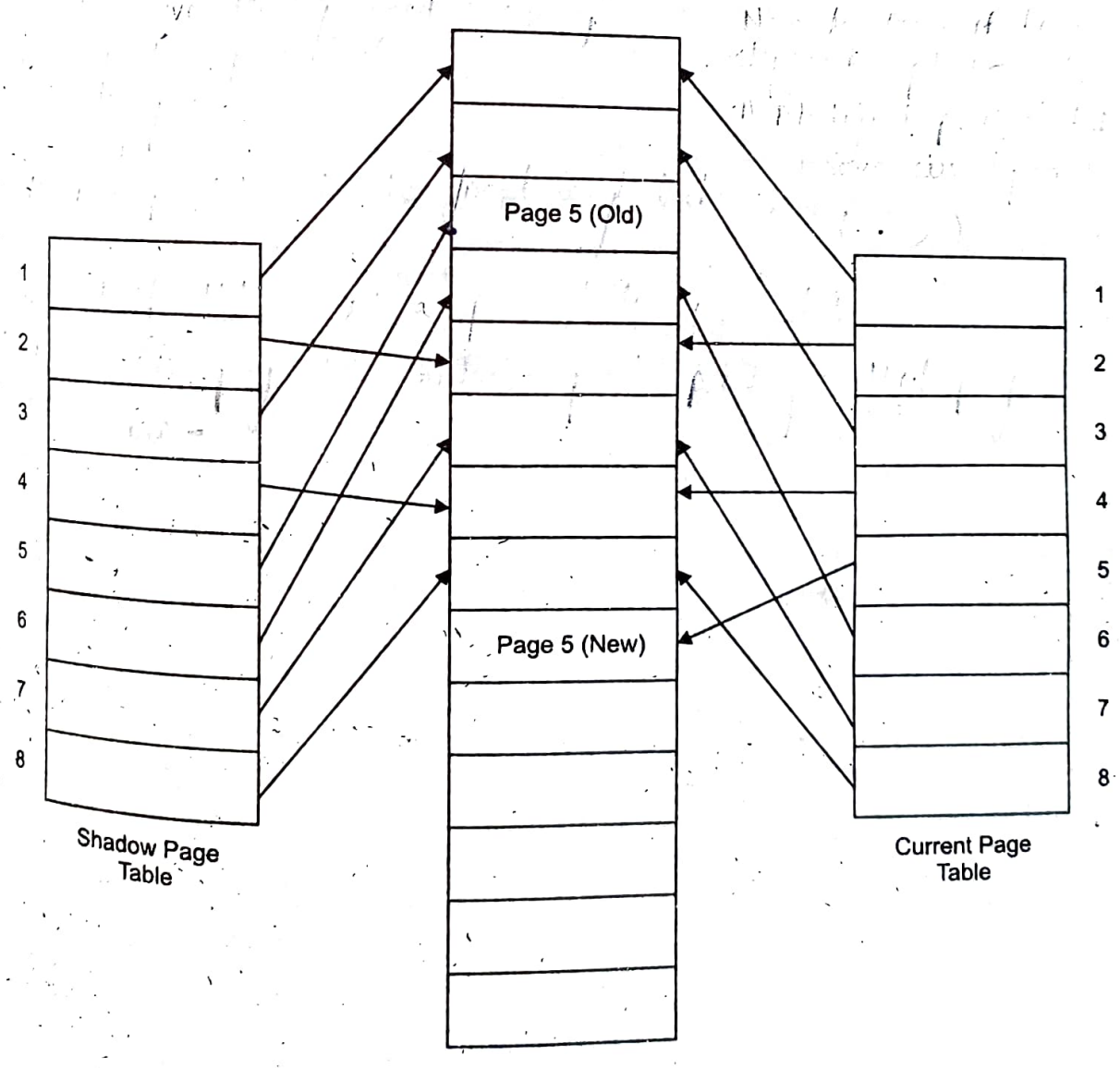

FIGURE 10.6. *Shadow paging.*

After performing all the above operations, the following steps are needed to commit Ti.

1. Modified pages are transferred from main memory to disk.
2. Save current page table on disk.
3. Change current page table into shadow page table (by changing the physical address).

Shadow page table must be stored in non-volatile or stable storage because it is used at the time of recovery. Suppose the system crashed and you have to recover it. After every successful completion of transaction, current page table is converted into shadow page table. So, shadow page table has to be searched. For making the search simple, store shadow page table on fixed location of stable storage. No redo operations need to be invoked. Shadow page table and Current page table are shown in Figure 10.6 after write operation on page 5.

Disadvantages

1. Data fregmentation due to fixed sized pages.
2. Wastage of memory space..
3. Extra overhead at the time of commit (by transferring of page tables)
4. It provides no concurrency.