9. Transaction Processing

Transaction management:

- ☐ Transaction
- ☐ Concurrency control
- ☐ Recovery system

Transaction:

- ☐ A *transaction* is an execution of a user program, and is seen by the DBMS as a series or list of actions.
- ☐ The actions that can be executed by a transaction include reads and writes of database objects, whereas actions in an ordinary program could involve user input, access to network devices, user interface drawing, etc.(Or)
- ☐ It is defined as a collection of operations that form a single logical unit of work is called transaction.

It is a foundation for concurrent execution and recovery from system failure in a DBMS

Eg: Select, commit, create etc.

The ACID Properties: A DBMS must ensure four important properties of transaction to maintain data in the face of concurrent access and System failures.

- ☐ ACID is used to refer to these 4 properties.
- ☐ **A** tomicity: All actions in the transaction happen, or none happen.
- ☐ **C** onsistency: If each transaction is consistent, and the DB starts consistent, it ends up consistent.
- ☐ **I** solation: Execution of one transaction is isolated from that of all others.
- ☐ **D** urability: If a transaction commits, its effects persist, if system failure occurs.

Very valuable properties of DBMSs

- without these properties, DBMS's would be much less useful

Transaction access data using two operations.

- **Read(x):** which transfers data item x from DB to a logical buffer belonging to transaction that executed read operation.
- **Write(x):** which transfers data item x to DB from a logical buffer belonging to transaction that executed write operation.
- Eg: Transfer of 100/-Rs from account A to account B.

T1: Read (A)

A: =A-100;

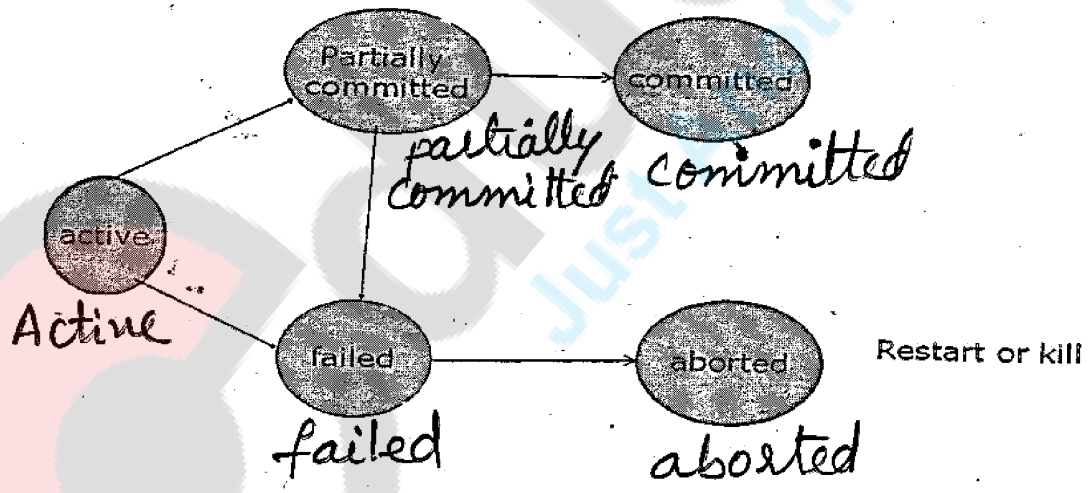
Write (A);

Read (B);

B: =B+100;

Write (B);

State diagram of Transaction



Write (C)

Tranasaction/D.S.Jagli

2

9.2

Atomicity of Transactions

- A transaction might commit after completing all its actions, or it could abort (or be aborted by the DBMS) after executing some actions.
- *Atomicity* means a transaction executes when all actions of the transaction are completed fully, or none are. This means there are no partial transactions.
- Atomic Transactions: a user can think of a transaction as always either executing all its actions, or not executing any actions at all.
  1. One approach: DBMS logs all actions so that it can undo the actions of aborted transactions.
  2. Another approach: Shadow Pages

Transaction Consistency

- "Consistency" - data in DBMS is accurate in modeling real world, follows integrity constraints.
- *Consistency* involves beginning a transaction with a 'consistent' database, and finishing with a 'consistent' database.
- For example, in a bank database, money should never be"created" or"deleted" without an appropriate deposit or withdrawal. Every transaction should see a consistent database.
- System checks ICs and if they fail, the transaction rolls back (i.e., is aborted).
  1. DBMS enforces some ICs, depending on the ICs declared in CREATE TABLE statements.
  2. Beyond this, DBMS does not understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).

Isolation

- Multiple users can submit transactions,Each transaction executes as if it was running by itself.

- Concurrency is achieved by DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.
- *Isolation* ensures that a transaction can run independently, without considering any side effects that other concurrently running transactions might have.

- Many techniques have been developed. Fall into two basic categories:
  - Pessimistic – don't let problems arise in the first place
  - Optimistic – assume conflicts are rare, deal with them *after* they happen.

Durability - Recovering From a Crash

- *Durability* defines the persistence of committed data: once a transaction commits, the data should persist in the database even if the system crashes before the data is written to non-volatile storage.
- System Crash - short-term memory lost
- Disk Crash - "stable" data lost
  - Need back ups; raid-techniques can help avoid this.
- At the end all committed updates and only those updates are reflected in the database.
  - Some care must be taken to handle the case of a crash occurring during the recovery process!

Concurrency Control:

Isolation and Serializability

Scheduling Transactions: A *schedule* is a series of (possibly overlapping) transactions.

- *Serial schedule:* A schedule that does not interleave the actions of different transactions.
  - i.e., you run the transactions serially (one at a time)

- *Equivalent schedules:* For any database state, the effect and output of executing the first schedule is identical to the effect of executing the second schedule.
- *Serializable schedule:* A schedule that is equivalent to some serial execution of the transactions.

  With a serializable schedule you only see things that could happen in situations where you were running transactions one-at-a-time.

Anomalies with Interleaved Execution

Conflict Serializable Schedules

- Two operations conflict if they are by different transactions, they are on the same object, and at least one of them is a write.
- Two schedules are conflict equivalent if They involve the same actions of the same transactions, and every pair of conflicting actions is ordered the same way Schedule S is conflict serializable if S is conflict equivalent to some serial schedule.

Some Examples of Conflicts

A *conflict* exists when two transactions access the same item, and at least one of the accesses is a write.

1. *lost update* problem

T: transfer ₹100 from *A* to *C*: *R(A)* *W(A)* *R(C)* *W(C)*

S: transfer ₹100 from *B* to *C*: *R(B)* *W(B)* *R(C)* *W(C)*

2. *inconsistent retrievals* problem (*dirty reads* violate consistency)

T: transfer ₹100 from *A* to *C*: *R(A)* *W(A)* *R(C)* *W(C)*

S: compute total balance for *A* and *C*: *R(A)* *R(C)*

3. *nonrepeatable reads*

T: transfer ₹100 from *A* to *C*: *R(A)* *W(A)* *R(C)* *W(C)*

S: check balance and withdraw ₹100 from *A*: *R(A)* *R(A)* *W(A)*

Legal Interleaved Schedules: Examples

$T < S$

1. Avoid *lost update* problem

   T: transfer \$100 from *A* to *C*: *R(A)* *W(A)* *R(C)* *W(C)*

   S: transfer \$100 from *B* to *C*: *R(B)* *W(B)* *R(C)* *W(C)*

2. Avoid *inconsistent retrievals* problem

   T: transfer \$100 from *A* to *C*: *R(A)* *W(A)* *R(C)* *W(C)*

   S: compute total balance for *A* and *C*: *R(A)* *R(C)*

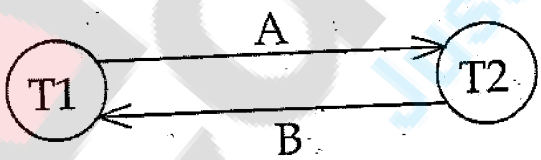
3. Avoid *nonrepeatable reads*

   T: transfer \$100 from *A* to *C* *R(A)* *W(A)* *R(C)* *W(C)*

   S: check balance and withdraw \$100 from *A*: *R(A)* *R(A)* *W(A)*

- **A schedule that is not conflict serializabl**
- The cycle in the graph reveals the problem. The cutput of T1 depends on T2, and vice-versa.

T1:	R(A), W(A),	R(B), W(B)
T2:	R(A), W(A), R(B), W(B)	



Dependency graph

9·6

Concurrent Executions: Two reasons for concurrency.

① Improved throughput and resourse utilization.

② Reduced waiting time.

Serial shedule: T_1 is followed by T_2

T_1	T_2
read (A)	
A := A-50	
write (A)	
read (B)	
B := B+50	
write (B)	
	read (A)
	temp = A*0.1
	A = A-temp
	write (A)
	read (B)
	B := B+temp
	write (B)

concurrent schedule:

T_1	T_2
read (A)	
A := A-50	
write (A)	
	read (A)
	temp = A*0.1
	write (A)

9.7

Serializability:

① data base system must control concurrent execution of transactions, to ensure that the db state remains consistent.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)

Transaction may perform an arbitrary sequence of operations on a data item i.e residing in the local buffer of the transaction.

The different forms of schedule equivalence

① conflict serializability.

② view serializability.

Conflict:

if I_1, I_2 are operations by different transactions T_1, T_2 on the same data item 'x', and at least one of these instructions is write operation then I_1, I_2 are conflict.

Conflict equivalent:

if schedule S_1 can be transformed into a schedule S_2 by series of swaps of non conflicting instructions, then S_1 and S_2 are conflict equivalent.

Eg:

S_1 =

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

=

S_2

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

① Conflict serializability:

S - Schedule

I_1, I_2 are instructions

T_1, T_2 are transactions

→ I_1, I_2 are using different data items then no problems or conflicts can arise.

→ I_1, I_2 are using same data item x then order of I_1, I_2 is matter

→ Four cases need to be consider.

① I_1 = read(x) I_2 = read(x)
order of I_1, I_2 doesn't matter for T_1 T_2

② I_1 = read(x), I_2 = write(x)
order of I_1, I_2 matters for T_1, T_2 execution

③ I_1 = write(x), I_2 = read(x)
order of I_1, I_2 matters for T_1, T_2 execution.

④ I_1 = write(x), I_2 = write(x)
order of I_1, I_2 directly affecting the final value of data item - x in schedule.

Conflict Serializable:-

The schedule S is conflict serializable if it is conflict equivalent to serial schedule.

View Serializability:-

Two schedules S1, S2, where the same set of transactions participates in both schedules.

The schedule S1, S2 are said to be view equivalent if following conditions are satisfied

1. for each data item X
   if T_1 reads intial value of x in schedule S1, then T_1 must in schedule S2 & read the intial value of X.
2. for each data item x.
   if T_1 executes read(x) in schedule S1, the value of x is produced by write(x) in T2. then read(x) of T_1 must be in schedule S2 value of x is same by write(x) in T2.
3. for each data item x.
   any transaction write final value of x in S1 must write final value of x in S2.

View Serializable :-

Schedule S is view serializable if it is view equivalent to a serial schedule.

Eg:

T3	T4	T5
read(x)		
write(x)	write(x)	
	write(x)	write(x)

S_1

T3	T4	T5
read(x)		
	write(x)	
write(x)		
		write(x)

S_2

View serializable schedule

Testing of Serializability :- (conflict)

designing concurrency control schemes, must show that schedules generated by scheme are serializable

- Determining conflict serializability by precedence graph.

$G = (V, E)$ V = all transactions in schedule
E = set of edges, all edges $T_1 \rightarrow T_2$ must hold one of three conditions

1. T_1 - write(x) before T_2 - read(x)
2. T_1 - read(x) before T_2 - executes write(x)
3. T_1 - write(x) before T_2 - write(x).

Eg:- $T_1 \rightarrow T_2$ $\qquad$ $T_2 \rightarrow T_1$

T_1	T_2
read(x)	
write(x)	
	read(x)
	write(x)

S1

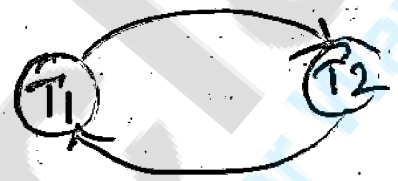
T_1	T_2
	read(x)
	write(x)
read(x)	
write(x)	

S2

Eg:

T_1	T_2
read(A)	
	read(A)
	write(A)
	read(B)
write(A)	
read(B)	
write(B	
	write(B)

precedence graph



① NO - cycle - Conflict & serializable

② cycle - Not conflict serializable

Serializability order of transactions can be obtained through topological sorting - which determines a linear order.

Testing view serializability is rather complicated.

Concurrency control Techniques: concurrency control Techniques that are used to ensure isolation property of concurrently executing transactions.

1. Locking Techniques
2. Time stamp techniques

1. Loking Techniques:

A lock is a variable associated with a data item describes the status of item with respect to possible operations that can be applied.

Lock protocol: set of rules to be followed by each transaction.

Binary Locks: It can have two states < lock, unlock >

Types of locks < Shared — read only — S, exclusive-write + read — X >

Lock conversions:

1. upgrade Lock S to X
2. down grade Lock X to S.

using binary locks or read/write locks in transactions, doesn't guarantee serializability.

TWO-PHASE LOCKING (2PL) :- (Basic)

It has two rules (2 phases)

① Expanding or growing phase :- In this phase new locks on items can be acquired but none can be released.

② shrinking phase: all existing locks can be released but no new locks can be aquired.

Eg:

T1	T2	T3	T4
S-lock(B)	S-lock(A)	S-lock(B)	S-lock(A)
read(B)	read(A)	read(B)	read(A)
unlock(B)	unlock(A)	X-lock(A)	X-lock(B)
X-lock(A)	X-lock(B)	unlock(B)	unlock(A)
read(A)	read(B)	read(A)	read(B)
A := A+100(y)	B := B+200(x)	A = A+B	B = B+A
write(A)	write(B)	write(A)	write(B)
unlock(A)	unlock(B)	unlock(A)	unlock(B)

S1 S2

① schedule S1 is non serializable schedule That uses locking, NO 2PL.

② Schedule S2 is guaranteeing serializability by 2PL but leads to deadlock.

9:15

There are a number of variations of 2PL.

1. Basic 2PL
2. Conservative 2PL (or) static 2PL
3. Strict 2PL
4. Rigorous 2PL

(2) static 2PL :- It requires a transaction to lock all the items it access before the transaction begins execution, by predeclaring its read-set and write-set.

- if any of the predeclared items needed cannot be locked, the transaction doesn't lock any item, it waits until all the items are available for locking.

Advantage: It is deadlock free

Disadvantage: Difficult to use in practice.

(3) strict 2PL:

- The most popular 2PL
- It guarantees strict schedule.
- Transaction doesn't release any of its X-locks until after it commits or aborts.
- leads to deadlock
- strict schedule for recoverability.

9.16

④ Rigorous 2PL:

- Transaction doesn't release any of its locks (S & X)
- guarantees strict schedule.
- easy to implement.
- leads to deadlock.

Deadlock Handling:

deadlock occurs when each transaction is waiting for some item that is locked by some other transaction

Eg.

T_1	T_2
S-lock(A)	
read(A)	
	S-lock(B)
	read(B)
X-lock(B)	
	X-lock(A)

wait for graph

① Deadlock prevention:

② Deadlock detection:

① Deadlock prevention:- The protocol which is used in conservative 2PL, requires lock all the items it needs in advance, if any of the items cannot be obtained, none of the items are locked. The transaction waits then tries again to lock all the items.

It limits concurrency.

deadlock prevention protocol with time stamp TS(T)

The timestamps are typically based on the order in which transactions are started.

The rules

① wait-die ② wound-wait

① wait-die:

TS(T1) < TS(T2) T1 starts before T2

T1 is older T2 is younger

TS is less for T1 & more for T2.

- if TS(T1) < TS(T2) then T1 is allowed to wait, otherwise T1 abort (die) and restart it later with same TS.

TS(T1) > TS(T2)

② wound-wait:

if TS(T1) < TS(T2) then abort T2 (wounds) and restart with same TS; otherwise T1 is allowed to wait.

- both the cases younger is aborting
- dead lock free techniques.

Deadlock detection & Recovery

Deadlock can be detected by creating a directed graph called wait-for graph.

A deadlock exists in the system if and only if the wait-for graph contains cycle.

Deadlock Recovery:

needs to recover the system from deadlock by performing the following actions:

① Select a victim: To determine transaction that will be rolled back to break the deadlock. The one to get selected should acquire minimum cost factors influence minimum cost are

- How long the Transaction has compute
- How many data items the transaction has used.
- How many data items needed.
- How many transaction will be is roll back.

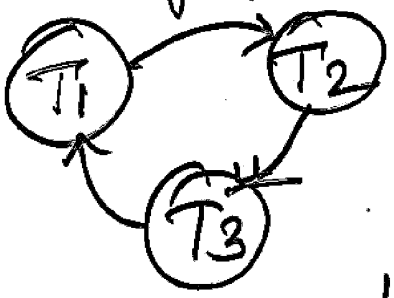
② Roll back: to determine how much should the victim transaction get rolled back.

- completely or partially.

③ Starvation: If ensure that same victim not to selected repeatedly.

Deadlock Detection:-

The system checks if a state of deadlock actually exists, by wait-for graph it can be detected.



wait-for graph.

(2) Concurrency control based on timestamp ordering

Timestamps: assigned TS in order in which the transactions are submitted to system.

- achieve dead lock-free without locks.
- each data item x can have two timestamp values.

(a) Read_TS(x):

(b) write_TS(x):

(a) read timestamp: This is the largest timestamp among all timestamps of transactions that have read data item x successfully.

(b) write timestamp: This is the largest TS among all TS of transactions that have written x.

The time stamp ordering protocol:

This is ensuring that any conflicting read & write operations are executed.

This operates as follows

① Suppose Transaction T_1 issues read(x)

ⓐ if $TS(T_1) < write\text{-}TS(x)$

then T_1 is rolled back since value of x is overwritten by a younger transaction. T_1 restarts.

otherwise T_1 performs read(x) & updates the Read-time stamp, to $TS(T_1)$.

② Suppose Transaction T_1 issues write(x)

if $TS(T_1) < Read\text{-}TS(x)$ or $write\text{-}TS(x)$

Then T_1 is rolled back.

otherwise T_1 performs write(x) and updates write-TS(x) to $TS(T_1)$.

Advantage

- It ensures serializability
- It also ensures freedom from deadlock.
- No waiting for any transaction.

Thomas write rule:

The modification to Time stamp ordering protocol, called thomas write rule.

① Read Rule is same as basic TO.

② write Rule i.e T_1 issues write(x)

- if TS(T_1) < Read_TS(x), then T_1 is rolled back.

Checks condition

if TS(T_1) < write_TS(x) then T_1 neither performs write operation nor is rolled back. T_1 writes an absolate value of x. (out dated value).

otherwise system executes write operation by T_1 and sets write_TS(x) to TS(T_1).

eg

T_1	T_2
read(x)	
	write(x
write(x	

Granularity of Data Item:

A database item could be choosen to be one of the following:

1. A Database record
2. A field value of a database record
3. A disk block
4. A whole file
5. The whole Database.

The Granularity can affect the performance of concurrency control and recovery.

Granularity: The size of data item.

Granularity < fine Granularity — Small size
Granularity < coarse Granularity — large size

Coarse granularity:-

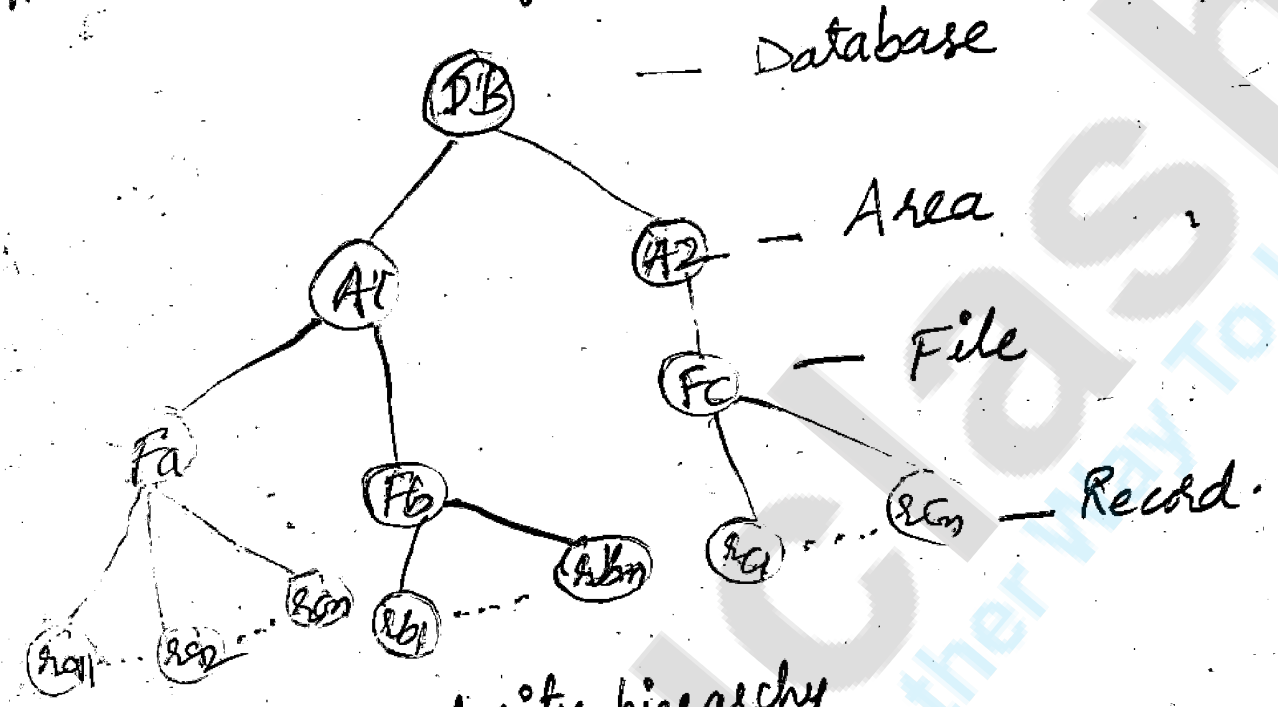
The larger the data item size, the lower the degree of concurrency.

Fine granularity:-

The smaller the data item size, the higher the degree of concurrency. but more lock & unlock operations will be performed, causing a higher overhead and more storage space is required for lock table.

Multiple Granularity:

Another specialized locking strategy, called multiple granularity locking allows us set locks on objects that contain other objects.



Granularity hierarchy

To make multiple granularity level locking practical additional types of locks, called intention locks are needed.

The idea behind intention locks is for a transaction to indicate, along the path from root to the desired node, what type of locks it will require from one of the node's descendants.

Types of intention locks: ① IS, ② IX ③ SIX

- Intention shared, Intention exclusive & shared Intention exclusive.

Compatibility Matrix:

	IS	IX	S	SIX	X
IS	True	T	T	T	F
IX	T	T	F	F	F
S	T	F	T	F	F
SIX	T	F	F	F	F
X	F	F	F	F	F

→ To lock a node in IS (respectively X) mode a transaction must first lock all its ancestors in IS (respectively IX) mode.

The multiple-granularity locking protocol:-

It ensures serializability.

The MGL protocol consists of the following rules:

① The lock compatibility must be adhered to

② The root of the tree must be locked first, in any mode.

③ A Node N can be locked by transaction T in S or IS mode only if the parent node N is already locked by transaction T in either IS or IX mode.

④ A Node N can be locked by transaction T in X, IX or SIX mode only if the parent in either IX or SIX.

⑤ It can lock a node only if it has not unlocked any node. (2PL)

⑥ Transaction T. unlock node N if child nodes are unlocked.

Recovery system of Transaction failure

Failure classification:

① Transaction failure:
- logical error
- system error

Log-Based Recovery:-

The most widely used structure for recording database modifications is the log. The log is a sequence of log records, recording all the update activities in the database.

There are several types of log records.
- insert log record, start, commit, abort.
- update log record.
- Delete log record.

An update log record describes single database write. It has fields

Tid	Data-Item id	old value	New value

Represent
<Ti start>; <Ti, x, V1, V2>
<Ti commit>; <Ti abort>

There 3 techniques

① Defferred Data base modifications

② Immediate Data base modification

③ check points.

whenever a transaction performs a write, it is essential that log record for that write be created before the database is modified.

① Deferred Database modification:

This Technique ensures transaction atomicity by recording all database modifications in the log, but deferring the execution of all write operations of transaction untill the transaction partially commits.

Since failure may occur while the updating is taking place, all log records must written to the stable storage.

A = 1000, B = 1000 C = 700

Eg.

T_1	T_2	
read (A)	read (C)	<T_1 start>
A = A − 50	C = C − 100	<T_1, A, 950>
write (A)	write C	<T_1, B, 1050>
read (B)		<T_1, commit>
B = B + 50		<T_2, start>
write (B)		<T_2, C, 600>
		<T_2, commit>

Redo (T_1) sets new values

9.27

② Immediate Database modification

- This Technique allows database modifications to be output to the database while the transaction is still in the active state.
- Data modifications written by active transactions are called uncommitted modifications.
- In the event of crash or a transaction failure, the system must use the old-value fields of the log records.

Eg: <T0 start> ①
<T0, A, 1000, 950>
<T1, B, 2000, 2050> ①
<T1, Commit>
<T2 start>
<T2, C, 700, 600> ②
<T2, commit> ②

Using the log, the system can handle any failure that doesnot result in the loss of information in stable storage.

The recovery scheme uses two procedures.

① Undo (Ti) restores the value of all data items updated by Ti to the old value.

② Redo (Ti) sets of the value of all data items updated by Ti to the new values.

9.28

3. Checkpoints :-

When a system failure occurs, we must refer the log to determine those transactions that need to be undo & redo.

There are 2 difficulties with searching log.

① Time consuming

② more redo operations (longer time for recovery).

To reduce overhead, checkpoint is introduced.

During the execution, the system maitains the log, in addition periodically performs checkpoints.

Sequence of actions in checkpoint Technique

① output onto stable storage all log records currently residing in main memory

② output the disk all modified buffer blocks

③ output onto stable storage a log record <checkpoint>

Eg: <T1, start>
<T1, A, 1000, 950> → ①
<T1, B, 2000, 2050>
<T1 commit> <check point> → ②
<T2, start>
<T2, C, 700, 600> → ③
<T2 commit> → ④

Advanced Recovery Techniques:

If DBMS gets crash, we can use ARIES recovery algorithm, which is simple, works well with a wide range of concurrency control mechanism & is used ~~to~~ an increasing no. of database systems.

ARIES:

It is designed to work with a steal, no-force approach.

After crash, restart proceeds in 3 phases.

(1) Analysis: Identify dirty pages in the buffer pool and active transactions at the time of crash.

(2) Redo: Repeat all actions, starting from an appropriate point in the log, restore the database.

(3) Undo: undoes the actions of transactions that did not commit.

Execution History

Eg:

10	update: T1 writes P5
20	update: T2 writes P3
30	T2 commit
40	T2 end
50	update T3 writes P1
60	update T3 writes P3
	* crash, Restart

Analysis:

Redo :

undo :

9.30

Shadow paging

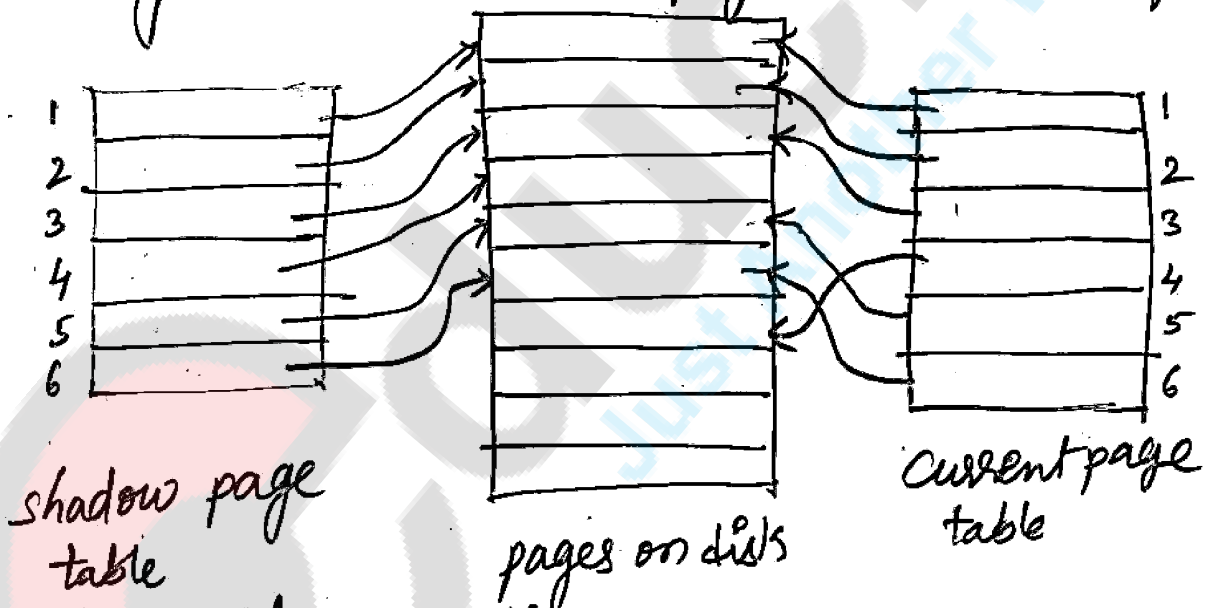
- Shadow paging is an alternative to log-based recovery.
- This is useful if transactions executes serially.

Idea:

→ Maintain two pages tables during the lifetime of a transaction – The current page table & shadow page table.

→ Store the shadow page table in nonvolatile storage i.e the state of the database prior to transaction execution may be recovered.

→ during execution shadow page is not modified.



→ **To commit Transaction**

1. Flush all modified pages to disks
2. output current page table to disk
3. make the current page table, the new shadow page table
4. once pointer to shadow page table has been written transaction is committed.

Advantages :-

① No overhead of writing log records.

② Recovery is trivial

Disadvantages:

① Copying the entire page table is very expensive.

② Data gets fragmented.

③ After every transaction completion, the database pages containing old versions of modified data need to be garbage collected.

④ Hard to extend to allow transactions to run concurrently.

WAL : write-ahead log protocol :-

Before writing a page to disk, every update log record that describes a change to this page must be forced to stable storage.

