DBMS

MODULE 6

Transaction Management,Concurrency and Recovery

Written & Compiled by,
Prof.Priyanka Bhilare

6. Transaction Management and Concurrency.

* Transaction Concept :-

A transaction is a unit of work i.e. performed against a DB. Transactions are units or sequences of work accomplished in a logical order whether in a manual fashion by a user or automatically by some sort of DB program.

Transactions are initiated by a user programs written in a high level data manipulation language or any programming language.

* Operations of Transactions :-

Following are the ~~two~~ three main operations performed on transactions :-

1) **Read (X) :-** Read operation is used to read the value of 'X' from the DB and stores it in a buffer in main memory.

2) **Modify (X) :-** It changes the value of item 'X' in the main memory.

3) **Write (X) :-** It is used to write the modified value from main memory to storage.

Scanned by CamScanner

For eg:-

Consider the transaction of withdrawing Rs. 50 from account A and adding Rs. 100 to account to B. We will consider Read & write operation for reading and writing the DB.

T1 :-

```
Read(A);
A := A-50;
write(A);
Read(B);
B := B+100;
write(B);
```

* Transaction States :-

A transaction in DB can be in one of the following states.

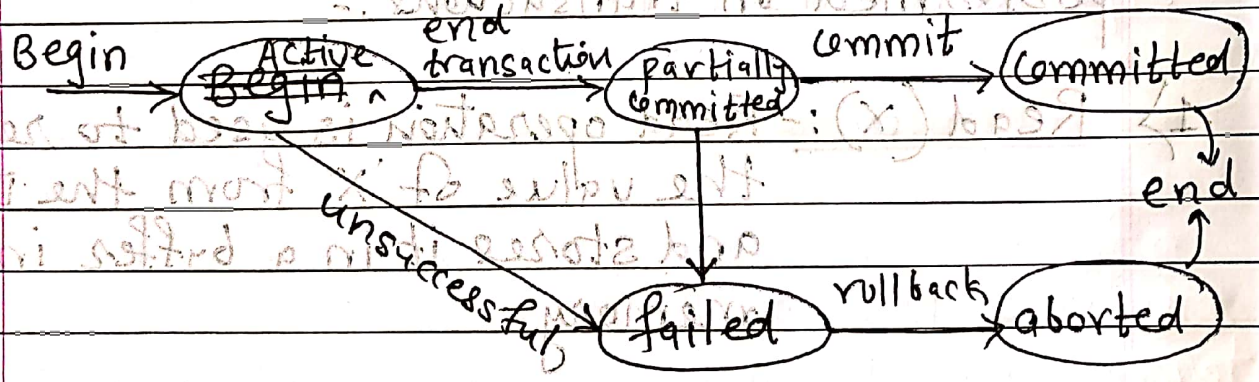


fig. States of Transaction

Scanned by CamScanner

- **Begin Transaction** :- A marker that specifies start of transaction execution.
- **End Transaction** :- A marker that specifies end of transaction.
- **Commit** :- A signal to specify that the transaction has been successfully completed and will not be undone.
- **Rollback** :- A signal to specify that the transaction has been unsuccessful and so all temporary changes in the DB are undone. A committed transaction cannot be undone.

1) **Active** :-
In this state transaction is being executed. This is the initial state of every transaction.

2) **Partially Committed** :-
The transaction enters this state after the last statement of the transaction.

3) **Committed** :-
The transaction enters this state after successful completion of the transaction and system checks have issued commit signal.

4) **Failed** :-
The transaction goes from partially committed state or active state to failed state when it is discovered that normal execution can no longer proceed or system checks fail.

Scanned by CamScanner

5) **Aborted :-**

This is the state after the transaction has been rolled back after failure and the DB has been restored to its state that was before the transaction began.

If any of the checks fail and the transaction has reached to failed state the recovery manager rolls back all its write operations on the DB to bring the DB back to its original state where it was prior to the execution of the transaction. Transaction in this state are called aborted. The DB recovery module can select one of the two operations after a transaction aborts :-

a) **Restart :-**

It can restart the transaction only if the transaction was aborted due to some hardware or software errors that were not created through the internal logic of transaction. A restarted transaction is considered to be a new transaction.

b) **Kill the transaction :-**

The transaction was killed because the input was bad or desired data was not found in the DB.

Scanned by CamScanner

* Transaction Properties :- (ACID properties)

To ensure consistency and integrity of data, it is very important for DB systems to maintain the following properties.

1) Atomicity :-

Let Ti be the transaction that transfers Rs. 50 from account A to account B. This transaction can be defined as

Ti : Read(A)
A := A - 50;
write (A);
Read (B);
B := B + 50;
write (B);

Atomicity says : "Either all operations of the transaction are reflected properly in the DB, or none are."

Suppose that, just before the execution of transaction Ti, the values of accounts A and B are Rs. 1000 and Rs. 2000 respectively. Now suppose that, during the execution of transaction Ti, a failure occurs that prevents Ti from completing its execution successfully.

Failure happened after the write(A) operation but before write(B) operation. In

Scanned by CamScanner

this case, the values of accounts A and B reflected in the DB are Rs. 950 and Rs. 2000. The system destroyed Rs. 50 as a result of this failure. The sum A+B is no longer preserved. We term such a state an inconsistent state.

This state however replaced by the consistent state, where the value of account A is Rs. 950 and the value of account B is Rs. 2050

The basic idea behind ensuring atomicity is this: The DB system keeps track of the old values of any data on which transaction performs a write and if transaction does not complete its execution, the DB system restores the old values to make it appear as though the transaction never executed.

Ensuring Atomicity is the responsibility of the DB system itself; specifically it is handled by a component called the transaction management component.

Consistency :-

Consistency says "Execution of a transaction in isolation (i.e. with no other transaction executing concurrently) preserves the consistency of the database."

The consistency requirement here is that the sum of A and B be unchanged by the execution of the transaction. Without the consistency requirement,

Scanned by CamScanner

money could be created or destroyed by the transaction. It can be verified easily ~~by~~ that, if the DB is consistent before an execution of the transaction, the DB remains consistent after the execution of the transaction.

Ensuring consistency for an individual transaction is the responsibility of the application programmer who codes the transaction.

3) Isolation :-

Isolation says "Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions Ti and Tj, it appears to Ti that either Tj finished execution before Ti started or Tj started execution after Ti finished. Thus each transaction is unaware of other transactions executing concurrently in the system."

The DB is temporarily inconsistent while the transaction to transfer funds from A to B is executing with the deducted total written to A and the increased total yet to be written to B. If a second concurrently running transaction reads A and B at this intermediate points and computes A+B, it will observe an inconsistent value. Furthermore, if this second transaction then performs updates on A and B based on the inconsistent values that it read, the DB may be left in an inconsistent state even after both transactions have completed.

Scanned by CamScanner

A way to avoid problem of concurrently executing transactions is to execute transactions serially - i.e. one after the other.

Ensuring the isolation property is the responsibility of a component of the DB system called the concurrency control component.

4) Durability :-

Durability says "After a transaction completes successfully, the changes it has made to the DB persist, even if there are system failures."

Ensuring durability is the responsibility of a s/w component of the DB system called the recovery-management component.

* Concurrent Execution

The transaction processing system allows multiple transactions to run concurrently. This causes several complications with the consistency of the data.

To avoid the complications it is better to run transactions serially.

The reasons for allowing transactions concurrently are :-

1) Improved throughput and Resource Utilization :-

A transaction consists of many steps.

Scanned by CamScanner

Some occupies I/O activities and some occupies CPU activity. The CPU and disk in computer system can operate in parallel. This can be used to run multiple transactions parallely. When one transaction is performing read and write operation on disk, another transaction can be running in the CPU. This increases the processor and disk utilization.

2) Reduces waiting time of Transactions :-

There may be some short and some long transactions running on a system. If the transactions are running serially, the other transactions have to wait for the previous one to finish. If the transactions are accessing diff^n DB, it is better to run them concurrently. But if they are accessing same DB then it is difficult. So when transactions are running concurrently the interaction among transactions must be maintained to control data consistency.

- Serial schedule :-

A serial schedule consists of set of transactions that are executing serially or concurrently.

A schedule represents the chronological order in which the instructions are executed in the system. i.e. execution sequences of transactions.

For example, consider the transaction T1 and T2 that transfer funds from account A to account B.

Scanned by CamScanner

- The transaction T1 is shown below:-

```
T1 : Read(A);
     A := A-50;
     write(A);
     Read(B);
     B := B+50;
     write(B);
```

- The transaction T2 is shown below:-

```
T2 : Read(A);
     A := A-100;
     write(A);
     Read(B);
     B := B+100;
     write(B);
```

Suppose the current value of A is 500 and the current value of B is 1000. The two transactions are executing in the sequence T1 followed by T2. We want to preserve the addition A+B.

-

T1	T2	
		A = 500
Read(A);		A = 500 - 50 = 450
A := A - 50;		
write(A);		A = 450
read(B);		B = 1000
B := B+50;		B = 1000+50 = 1050
write(B);		B = 1050 (A+B = 1500 is preserved)
	Read(A);	A = 450
	A := A - 100;	A = 450 - 100 = 350
	write(A);	A = 350
	Read(B);	B = 1050

Scanned by CamScanner

T_1	T_2	
	$B := B + 100;$	$B = 1050 + 100 = 1150$
	$write(B);$	$B = 1150$
		($A + B = 1500$ is preserved)

fig:- Serial schedule 1 (T_1 followed by T_2).

T_1	T_2	
	$Read(A)$	$A = 500$
	$A := A - 100;$	$A = 500 - 100 = 400$
	$write(A);$	$A = 400$
	$Read(B);$	$B = 1000$
	$B := B + 100;$	$B = 1000 + 100 = 1100$
	$write(B);$	$B = 1100$
$Read(A);$		$A = 400$
$A := A - 50;$		$A = 400 - 50 = 350$
$write(A);$		$A = 350$
$Read(B);$		$B = 1100$
$B := B + 50;$		$B = 1100 + 50 = 1150$
$write(B);$		$B = 1150$
		($A + B = 1500$ is preserved)

fig. Serial schedule 2 (T_2 followed by T_1)

After changing the sequence the addition of $A + B$ is preserved in both the cases. So serial schedule 1 is equal to serial schedule 2.

Scanned by CamScanner

Concurrent schedule :-

In a multitransaction environment serial schedules are considered as benchmark. The execution sequence of an instruction in a transaction cannot be changed, but two transactions can have their instructions executed in a random fashion. This execution does no harm if two transactions are mutually independent and working on different segments of data, but in case there are 2 transactions working on the same data then the results may vary. This ever varying result may bring the DB to an inconsistent state

To resolve this problem, we allow parallel execution of a transaction schedule.

When the multiple transactions are executing concurrently in the system the schedule is said to be concurrent schedule. If two transactions are executing concurrently then O.S. will execute first transaction then perform a context switch and will execute second transaction and will again switch back to first.

Scanned by CamScanner

T_1	T_2		
Read(A);		A = 500	
A := A − 50;		A = 500 − 50	
write(A);		A = 450	
	Read(A);		A = 450
	A := A − 100;		A = 450 − 100
	write(A);		A = 350
Read(B);			
B := B + 50;		B = 1000	
write(B);		B = 1000 + 50	
	Read(B);	B = 1050	
	B := B + 100;		B = 1050
	write(B);		B = 1050 + 100
			B = 1150

fig. Concurrent schedule 1.

The o/p above schedule is equivalent to serial schedule (as A+B=1500 is preserved) that leads the DB in consistent state.

T_1	T_2
Read(A); 500	
A := A − 50; 450	
	read(A); 500
	A := A − 100; 400
	write(A); 400
	read(B); 1000
write(A); 400	
read(B); 1000	
B := B + 50; 1050	
write(B); 1050	

Scanned by CamScanner

T_1	T_2	
	B := B + 100;	— 1150
	write(B);	— 1150

fig. Concurrent schedule 2.

If we try to run the above schedule concurrency is not provided as (A+B=1500) is not preserved in this case.

Note :- If we have serial schedule consistency is always provided; but if we have concurrent schedule consistency may or may not be provided.

* Serializability in DBMS :-

- When multiple transactions run concurrently then it may give rise to inconsistency of DB.
- Serializability is a concept that helps to identify which non-serial schedules are correct & will maintain the consistency of DB.

• Serializable schedule :-

If a given non-serial schedule of 'n' transactions is found to be equivalent to some serial schedule of 'n' transactions, then it is called as 'serializable schedule'.

Scanned by CamScanner

- Difference between serial schedule and Serializable schedule :-

The only difference between serial schedule and serializable schedule is that, in serial schedule, only one transaction is allowed to execute at a time i.e. no concurrency is allowed.

Whereas in serializable schedule, multiple transactions can execute simultaneously i.e. concurrency is allowed.

Serializable schedules are always better than serial schedules as they improve both resource utilization and CPU through

- Types of Serializability :-

1) Conflict serializability
2) View serializability

1) Conflict Serializability :-

If a given schedule can be converted into a serial schedule by swapping its non-conflicting operations then it is called as Conflict serializable schedule.

- Conflicting operations :-

Two operations are called as conflicting operations if all the following

Scanned by CamScanner

Conditions hold true for them :-

1) Both the operations must belong to different transactions.
2) Both the operations must on same data item.
3) At least one of the two operations is a write operation.

- **Conflict equivalent :-**

If we can transform schedule 'S' into schedule 'S' by series of swaps of non-conflicting instructions we say that S and S' are conflict equivalent.

eg :- Consider the foll. schedule

T_1	T_2
$R_1(A)$;	
$W_1(A)$;	
	$R_2(A)$;
$R_1(B)$;	

In this schedule $W_1(A)$ and $R_2(A)$ are called as conflicting operations because all the above conditions hold true for them.

Scanned by CamScanner

(*) Concurrent schedule be conflict serializable.

(1)

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)

swapped

(2)

T_1	T_2
R(A)	
W(A)	
	read(A)
R(B)	
	W(A)
W(B)	
	R(B)
	W(B)

Seagull Division
PARADIGM
Medvision Pvt. Ltd.

Schedule S is conflict

Scanned by CamScanner

(3)

T_1	T_2
R(A)	
W(A)	
R(B)	
	R(A)
	W(A)
W(B)	
	R(B)
	W(B)

(4)

T_1	T_2
R(A)	
W(A)	
R(B)	
W(B)	
	R(A)
	W(A)
	R(B)
	W(B)

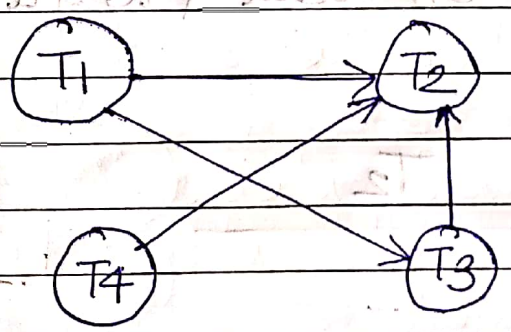
Scanned by CamScanner

eg:-

1) Check whether the given schedule 'S' is conflict serializable or not

T_1	T_2	T_3	T_4
			R(A)
	R(A)		
		R(A)	
W(B)			
	W(A)		
		R(B)	
	W(B)		

precedence graph



Since there exists no cycle in the precedence graph, therefore the given schedule S is conflict serializable.

Scanned by CamScanner

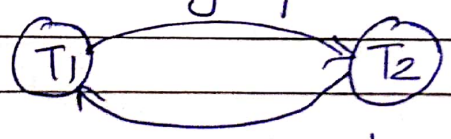
2)

T_1	T_2
R(A);	
A = A - 10;	
	R(A);
	Temp = 0.2 * A;
	w(A);
	R(B);
w(A);	
R(B);	
B = B + 10;	
w(B);	
	B = B + temp;
	w(B);

→ In this focus on read & write operations only, therefore

T_1	T_2
R(A);	
	R(A);
	w(A);
	R(B);
w(A);	
R(B);	
w(B);	
	w(B);

Precendence graph :-



Schedule is not conflict serializable as cycle is present.

Scanned by CamScanner

2) View serializability :-

If a given schedule is found to be view equivalent to some serial schedule then it is called as view serializable schedule.

- View equivalent schedule :-

Consider two schedules S_1 & S_2 each consisting of two transactions T_1 & T_2. schedules S1 & S2 are called view equivalent if the following three conditions hold true for them.

Condition 1 :-

If transaction T_i reads a data item X from the DB initially in schedule S1, then in schedule S2 also T_i must perform initial read of the data item X from the DB. The same is true for all data items.

Rule :-

Initial read must be same for all items.

Condition 2 :-

If a transaction reads a data item that has been updated by the transaction T_j in schedule S1 then in schedule S2 also transaction T_i must read the same data

Scanned by CamScanner

item that has been updated by transaction T_j.

Rule:-

Write-read sequence must be same.

Condition 3 :-

If data item x has been updated at last by transaction T_i in schedule S1, then in schedule S2 also the data item X must be updated at last by transaction T_i. The same has to be true for all data items.

Rule:-

Final writes must be same for all items.

eg:-

S1		S2	
T_1	T_2	T_1	T_2
R(A)		R(A)	
W(A)	R(A)	W(A)	
	W(A)	R(B)	
R(B)		W(B)	
W(B)	R(B)		R(A)
	W(B)		W(A)
			R(B)
			W(B)

Scanned by CamScanner

	A		B	
	S_1	S_2	S_1	S_2
1) First Read	T_1	T_1	T_1 ~~T_2~~	~~T_2~~ T_1
2) final write	T_2	T_2	T_2	T_2
3) Write-read	T_1	T_2	T_1	T_2

As it satisfies all 3 conditions therefore both the schedules are view equivalent.

* **How to check view serializability :-**

Method 1) If a given schedule is conflict serializable then it is surely view serializable.

Note :- All conflict serializable schedules are view ~~serializabor~~ serializable but all view serializable schedules may or may not be conflict serializable.

Method 2) No blind write (writing without reading a value is known as blind write)

rule :- No blind write means not a view serializable. If blind-write is there may or may not be view serializable.

Method 3) By using ~~three~~ conditions above write all dependencies and then draw a graph using those dependencies. If there ~~not~~ existed no cycle then schedule is view serializable otherwise not.

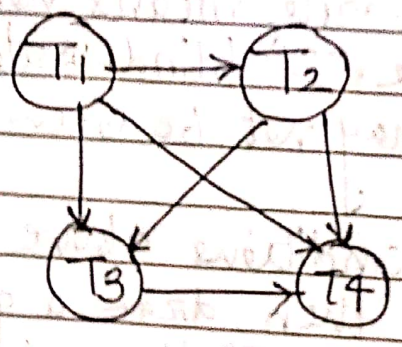
Scanned by CamScanner

(1) Check whether given schedule is view serializable or not.

T_1	T_2	T_3	T_4
R(A)			
	R(A)		
		R(A)	
			R(A)
W(B)			
	W(B)		
		W(B)	
			W(B)

(a) We know that if schedule is conflict serializable, then it is surely view serializable.

Precedence Graph



As no cycle is present it is conflict serializable and therefore it is view serializable also.

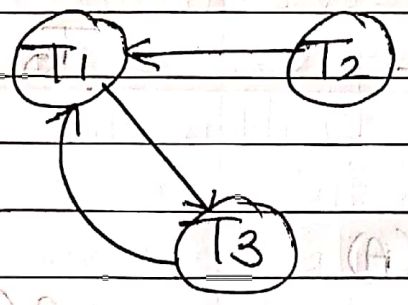
Scanned by CamScanner

(2)

T_1	T_2	T_3
R(A)		
	R(A)	
		W(A)
W(A)		

a) If it is conflict serializable then it is view serializable too.

<u>Precendence graph</u>



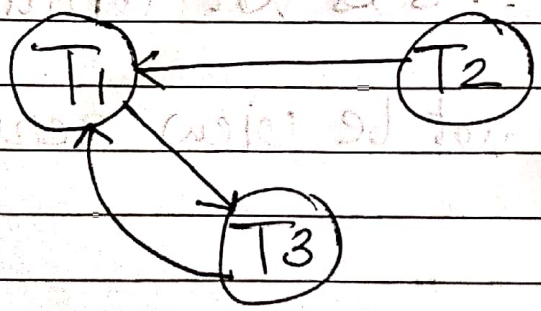
exists a cycle therefore not conflict serializable.

∴ may or may not view serializable.

b) checking blind-writes :-

Exists blind-write $W_3(A)$ therefore may or may not be view serializable.

c) Drawing dependency graph :-



exists a cycle in the dependency graph therefore schedule S is not view serializable

Scanned by CamScanner

i) Transaction T_1 firstly reads A and Trans. T_3 firstly updates A. Therefore T_1 must executed before T_3.

$\therefore$ We get dependency as $T_1 \longrightarrow T_3$.

ii) Final updation on A is made by Trans. T_1

$\therefore$ T_1 must execute after all other trans.

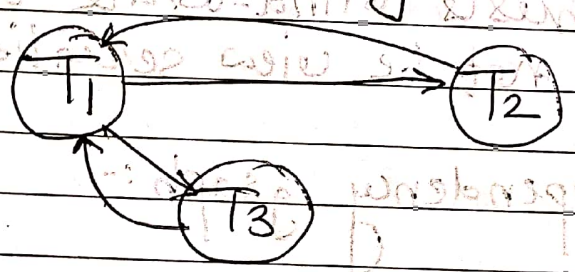
So get dependency $(T_2, T_3) \longrightarrow T_1$.

iii) There exists no-write sequence.

(3)

T_1	T_2	T_3
R(A)		
	W(A)	
		R(A)
W(A)		
		W(A)

a) Conflict serializability



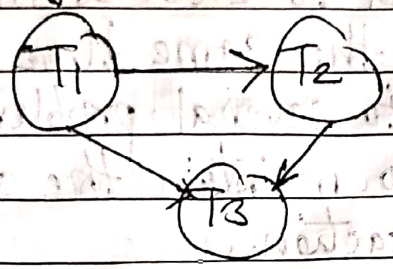
exists a cycle $\therefore$ S is not conflict serializable.

$\therefore$ may or may not be view serializable

6) <u>blind-write</u> :-

exists blind write. Therefore may or may not be view serializable.

Dependency graph



No cycle exists in the graph, therefore it is view serializable.

Scanned by CamScanner

* Concurrency Control :-

• Need of Concurrency Control

To make system efficient and to save time, it is required to execute more than one transaction at the same time; But Concurrency leads to several problems as it is highly important to maintain the order of execution of transaction.

• Problems of Concurrency Control

1) Lost Update Problem
2) Dirty Read
3) Inconsistent Retrievals problem.

1) Lost Update Problem:-

If any transaction Tj updates any variable v at time t without knowing the value of v at time t then this may lead to lost update problem. Consider the transaction shown below

Transaction Ti	Time	Transaction Tj
read(v)	t1	
	t2	read(v)
update(v)	t3	
	t4	update(v)

fig. The lost Update Problem

Scanned by CamScanner

At time t1 and t2, transactions ti and tj reads variable v respectively and get some value of v; at time t3, ti updates v but, at t4, tj also updates v (old value of v) without looking the new value of v (updated by ti). So, updation made by ti at t3 is lost at t4 because Tj overwrites it.

2) <u>Dirty Read</u>

The dirty reads occur in the case where one transaction updates an item of the DB and then the transaction fails for some reason the updated data item is accessed by another transaction before it is changed back to the original value.

Transaction Ti	Time	Transaction Tj
———	t1	———
———	t2	update (A)
read (A)	t3	———
	t4	rollback
———	t5	———

fig. Dirty Read

At time t2, transaction Tj writes A's value. At time t3, trans. Ti reads A's value. At time t4 trans. Tj rollbacks, so it changes A's value back to that of prior to t1. So trans. Ti now contains a value which

Scanned by CamScanner

has never become part of the stable DB. Such a problem is known as dirty-read problem as one trans. reads a dirty value which has not been committed.

3) <u>Inconsistent Retrieval Problem :-</u>

Consider the foll. transaction :-

T_i	Time	T_j
Read(A)=10	t1	
read(B)=20	t2	
	t3	write(A) A=50
	t4	commit
add A,B	t5	
A+B=30 not 60.		

Transaction T_i reads variable A & B at time t1 & t2 respectively, but at time t3, t_j updates value of A from 10 to 50 & commits at t4 that makes changes permanent. So addition of A & B at t5 gives wrong result. This leads inconsistency in DB because of inconsistent analysis by T_i.

Scanned by CamScanner

Concurrency control in DBMS:-

When more than one transactions are running simultaneously there are chances of conflict to occur which can leave DB to an inconsistent state. To handle these conflicts we need concurrency control in DBMS, which allows transactions to run simultaneously, but handles them in a such way that the integrity of data remains intact.

eg:- You and your brother have joint bank account from which you both can withdraw money. Now lets say you both go to different branches of the same bank at the same time and try to withdraw 5000 Rs. Your joint account has only 6000 balance. Now, if we don't have concurrency control in place you both can get 5000 INR at the same time but once both the transactions are finished the acc. balance will be -4000 which is not possible and leaves the DB in inconsistent state.

Therefore, we need something that controls the transactions in a such way that it allows transactions to run concurrently but maintaining the consistency of data to avoid such issues.

Solution to such conflict is lock.

Scanned by CamScanner

* Locks :-

To maintain consistency and to restrict modification of the data items by any other transaction which is presently accessed by some transactions. These restrictions can be applied by applying locks on data items. To access any data item, trans. have to obtain lock on it.

A lock is a kind of mechanism that ensures the integrity of data. Therefore two types of locks.

1) **Shared Lock (S) :-**

Shared lock is placed when we are reading a data. multiple shared locks can be placed on data but when shared lock is placed no exclusive lock can be placed.

eg :- When two transactions are reading steve's account balance. Let them read by placing shared lock but at the same time if another transaction wants to update the steve's account balance by placing exclusive lock ~~to~~ do not allow it untill reading is finished.

Scanned by CamScanner

2) Exclusive Lock (X) :-

Exclusive lock is placed when we want to read and write the data. This lock allows both the read and write operation. Once this lock is placed on data no other lock (shared or Exclusive) can be placed on the data untill Exclusive lock is released.

eg:- When a trans. wants to update the steve's account balance, Let it do by placing 'X' lock on it but if second transection wants to read the data ('S' lock) dont allow it, if another transaction wants to write the data (X-lock) don't allow that either.

so based on these we created a table like below:-

	S	X
S	True	false
X	false	false

Table:- Lock Compatibility Matrix.

How to read the matrix:-

There are two rows. first row says that when 'S' lock is placed another 'S' lock can be acquired so it is marked true but no exclusive locks can be acquired

Scanned by CamScanner

So marked false.

In second row, when lock X is acquired neither 'S' nor 'X' can be acquired So both marked false.

eg:- Consider the banking system in which you have to transfer Rs. 500 from account A to account B. Transaction T1 in fig. shows transfer of amount and T2 shows sum of account A and B. Initially account A has 1000 Rs. and B has 300/-

T_1	T_2
X-lock(A);	S-lock(B);
read(A);	read(B);
A = A - 500;	unlock(B);
write(A);	S-lock(A);
unlock(A);	read(A);
X-lock(B);	unlock(A);
read(B);	display(A+B);
B = B + 500;	
write(B);	
unlock(B);	

fig. Transaction T_1 & T_2 for banking example.

Scanned by CamScanner

T_1	T_2
X-lock (A);	
read (A); — A = 1000	
A = A − 500, — A = 500	
write (A); A = 500	
unlock (A);	
	S-lock (B);
	read (B); — B = 300
	unlock (B);
	S-lock (A);
	read (A); — A = 500
	unlock (A);
	display (A+B); A+B = 800
X-lock (B);	
read (B); — 300	
B = B + 500; B = 800	
write (B); B = 800	
unlock (B);	

In the schedule shown in fig. above T_2 gives wrong result i.e. rs. 800. Instead of Rs. 1300; because unlock A is performed immediately after its final access.

So, the improvement is keep all locks until the end of transaction.

Scanned by CamScanner

The modified T_1 and T_2 are shown in fig. below :-

T_1	T_2
X-lock(A);	
read(A); A=1000	
A=A-500; A=500	
write(A); A=500	
X-lock(B);	
read(B); B=300	
B=B+500; B=800	
write(B); B=800	
unlock(A);	
unlock(B);	
	slock(B);
	read(B); B=800
	S-lock(A);
	read(A); A=500
	display(A+B); A+B=1300
	unlock(A);
	unlock(B);

Advantages:-

1) It provides serializability.
2) It solve all concurrency problems.

Disadvantage:-

1) It leads to deadlock problem.

Scanned by CamScanner

T_1	T_2
x-lock (A);	
read (A);	
A = A − 500;	
write (A)	
	S-lock (B);
	read (B);
	slock (A) ↓ not possible because A is locked exclusively by T_1 & T_2 has to wait for A
x-lock (B); ↓ not possible because B is locked in shared mode by T_2 & T_1 has to wait for B.	

* Two-phase locking protocol :-

There are two phases in 2-phase locking protocol.

1) Growing phase :- In the growing phase, a new lock on the data item may be acquired by the transaction, but none can be ~~released~~ released.

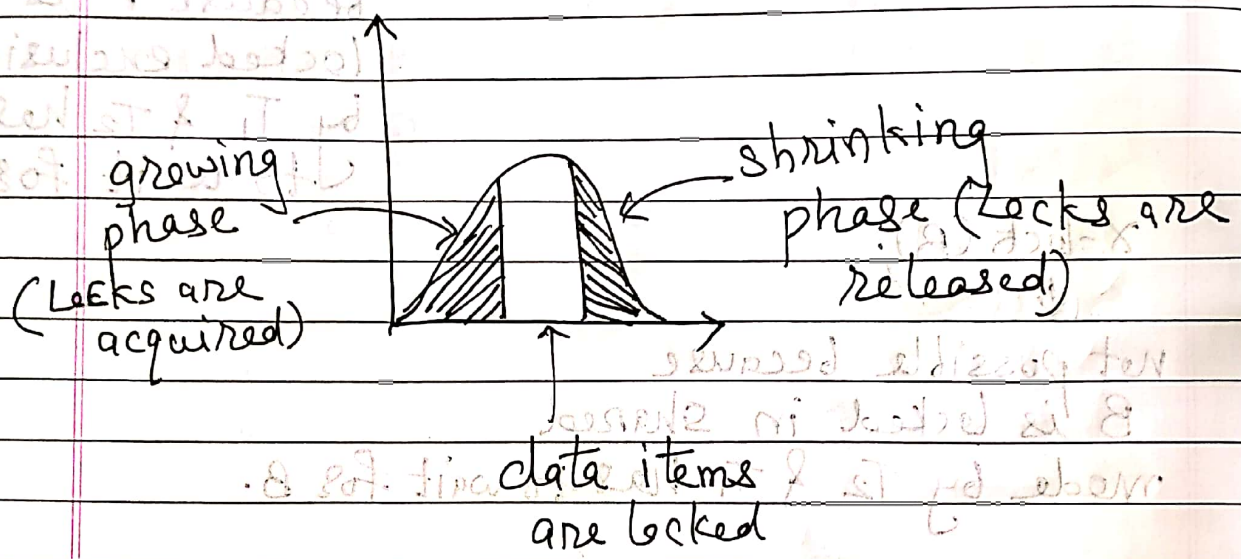
Scanned by CamScanner

2) Shrinking phase :-

In the shrinking phase, existing lock held by the transaction may be released, but no new locks can be acquired.

Lock point :-

In growing phase the point at which transaction has obtained its final lock is called lock point.



(*) Variations of Two phase Locking Protocol:

1) Strict 2PL protocol.
2) Rigourous 2PL protocol
3) Conservative 2PL protocol.

Scanned by CamScanner

1) Strict 2phase Locking Protocol (Strict 2PL)

In strict 2PL protocol, all exclusive locks are kept until the end of the transaction or until trans. commits (not bothered about shared lock).

S1

T3	T4
s-lock (A);	
read (A);	
X-lock (B);	
read (B);	
X-lock (C);	
unlock (B);	
read (C);	
write (C);	
	X-lock (B);
	read (B);
	write (B);
rollback	rollback due to rollback of T3.

fig. 2PL basic protocol.

In S1, there is a cascading rollback of T4 due to rollback of T3. but in S2 all exclusive locks are kept till end and avoid cascading rollback; but deadlock is present.

Scanned by CamScanner

T3	T4
s-lock (A);	
read (A);	
X-lock (B);	
read (B);	
write (B);	
s-lock (C);	
read (C);	
unlock (C);	
Commit;	
unlock (B);	
unlock (A);	
	X-lock (B);
	read (B);
	write (B);

fig. Strict 2PL.

2) <u>Rigorous 2phase Locking Protocol :-</u>

In Rigorous 2PL protocol, all locks are kept until the transaction commits. It means both shared and exclusive locks cannot be released till the end of the transaction. It does not cause cascading rollback but deadlock is still present.

Scanned by CamScanner

3) Conservative 2PL protocol :- (static 2PL)

Conservative 2PL protocol is also known as static 2PL protocol, this requires the transaction to look all the items it access before the transaction begins the execution by predeclaring its read-set and write set. The operation of data cannot start until we lock all the items required.

We can release the locks whenever we want but we need to lock all the data items before carrying out any operation. This makes it deadlock free, but cascading rollback is not ensured.

It is very difficult to use this protocol in practice because of need to pre-declare the read-set and write-set which is not possible in many cases.

T_1	T_2
X-lock (A);	
X-lock (B);	
read (A);	
A := A + 50;	
write (A);	X-lock (A);
unlock (A);	read (A);
read (B);	A := A + 50;
B := B + 100;	write (A);
write (B);	unlock (A);
unlock (B);	

Scanned by CamScanner

* Time-stamp ordering Protocol:-

1) Suppose transaction Ti issues read(Q)

i) If Ts(Ti) < w-timestamp (Q)
if yes read operation is rejected & Ti is rolled back.

ii) If Ts (Ti) ≥ w-timestamp (Q) then read operation accepted and R-timestamp (Q) = Ts (Ti).

2) Suppose Ti issues write (Q)

i) If Ts(Ti) < R-timestamp(Q)
OR
Ts(Ti) < w-timestamp (Q)
write request rejected, Ti is rolled ~~back~~ back.

ii) Otherwise write operation is executed and w-timestamp (Q) = Ts(Ti)

Scanned by CamScanner

(TS=10) T_1	TS(10.05) T_2	
①		RTS(A) = ~~0~~ ~~10~~ 10.05 RTS(B) = ~~0~~ ~~10~~ 10.05 WTS(A) = ~~0~~ 10.05
ⓐ read(B);		ⓐ WTS(B) = ~~0~~ 10.05 RTS(B) = TS(T_1) = 10
	ⓑ read(B);	ⓑ TS(T_2) > RTS(B) so read operation is performed ∴ RTS(B) = TS(T_2) = 10.05
	b := b−20; ⓒ write(B);	ⓒ WTS(B) = TS(T_2) = 10.05
ⓓ read(A);		ⓓ RTS(A) = TS(T_1) = 10
	ⓔ read(A)	ⓔ RTS(A) = TS(T_2) = 10.05
print(A+B)		
	A := A+30; ⓕ write(A);	ⓕ WTS(A) = TS(T_2) = 10.05
	print(A+B);	

<u>Note</u>:- From above schedule we find that both the transactions performed concurrently under the timestamp schedule without giving inconsistent results. Also note that this schedule ensures freedom from deadlock since no transaction ever waits.

Scanned by CamScanner

(2) $RTS(A) = 0$ $\quad$ $RTS(B) = \cancel{0}\ \cancel{10}\ 10.05$

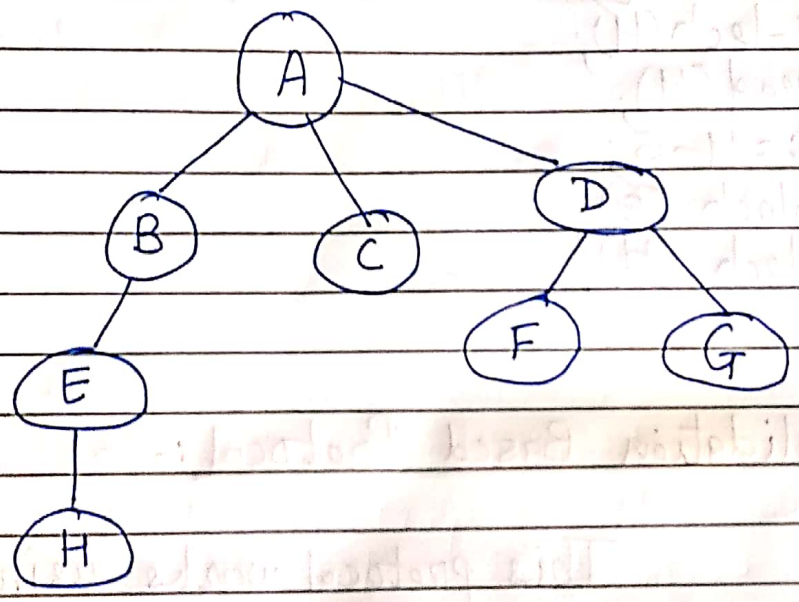
$WTS(A) = \cancel{0}\ 10.05$ $\quad$ $WTS(B) = \cancel{0}\ 10.05$

T1	T2
(a) read(B);	(a) $RTS(B) = TS(T_1) = 10$
	(b) $RTS(B) = TS(T_2) = 10.05$
(b) read(B); b := b − 20;	
(c) write(B);	(c) $TS(T_2) > WTS(B)$ ∴ $WTS(B) = TS(T_2) = 10.05$
(d) read(A);	(d) $TS(T_2) \geq WTS(A)$ ∴ $RTS(A) = 10.05$;
(e) a = a + 30; (e) write(A);	(e) $TS(T_2) < RTS(A)$ - false; $TS(T_2) < WTS(A)$ - false ∴ write operation is executed $WTS(A) = TS(T_2) = 10.05$
(f) read(A);	(f) $TS(T_1) < WTS(A)$ - rejected; $TS(T_1) \geq WTS(A)$ = accepted; $TS(T_1) < WTS(A)$ - true; $10 < 10.05$ ✓ ∴ read(A) is rejected and T1 is rolled back. Then T1 has to restart with new timestamp assigned to T1.

* Graph Based or Tree Based Protocol :-

Rules :-

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 A then T has to lock its parent first.
3) Transaction can unlock data items at any time.
4) Suppose 'T' has obtained lock on data item A and then release it after sometime then T cannot obtain any lock on A again.



We want to operate $G = G + 10$ and
$H = H - 5$

then,

Scanned by CamScanner

T1

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);

* Validation Based Protocol :-

This protocol works using 3 phases :-

1) **Read phase :-**

Transaction Ti reads all the data items and store them in temporary variable. After reading, all write operations are made on temporary variable instead of actual DB.

Scanned by CamScanner

2) Validation Phase :-

In validation phase validation test is performed to determine whether changes in actual DB can actually be made.

3) Write-phase :-

If Ti cleared the validation test then actual DB is changed according to the temporary variable. Validation test ensures violation free execution.

Three different timestamps associated with this protocol are :-

i) ~~Start~~ Execution (Ti) - Time when Ti finished read phase and starts validation phase.

ii) ~~Validation~~ start (Ti) - Time when Ti starts execution.

iii) finish (Ti) - Time when Ti finished with write phase.

The validation phase test for transaction (Ti) requires that all transactions (Tk) with Ts(Tk) < Ts(Ti) one of the following two conditions must hold:-

i) Finish (Tk) < Start (Ti). Since (Tk) completes its execution before Ti started, the serializability order is indeed maintained.

ii) The set of data items written by (Tk) does

Scanned by CamScanner

not interact with the set of data items read by Ti, and Tk completes its write phase before Ti starts its validation phase (start(Ti) < finish(Tk) < Validation (Ti)).

This condition ensures that the writes(Tk) and (Ti) do not overlap. Since the writes of Tk do not affect the read of Ti and since Ti cannot affect the read of Tk. The serializability order is indeed maintained.

T25	T26
read(B);	
	read(B);
	B := B − 50;
	read(A);
	A := A + 50;
read(A);	
<validate>	
display (A+B);	
	<validate>
	write(B);
	write(A);

Consider Transactions T25 and T26. Suppose that TS(25) < TS(T26) then, the validation phase succeeds. The write to the actual variables are performed only after the validation phase of T26. Thus, T25 reads the old values of B and A and this schedule is serializable.

Scanned by CamScanner

Recovery in DBMS

Failure :-

Failure refers to the state when system no longer continue with its normal execution that results in loss of information.

Different types of failures :-

a) System crash :- This failure occurs due to the bugs in the s/w, h/w failure.

b) Transaction failure :- This failure happens due to the logical errors such as i/p data missing, less free space, deadlock occurs.

c) Disk failure :- happens due to read-write head crashed or failure during transfer of data.

Recovery :-

Recovery from failure state refers to the method by which system restores DB to its most consistent state just before the time of failure.

There are three recovery techniques :-

i) Log Based Recovery
ii) Shadow Paging
iii) Checkpoints.

Scanned by CamScanner

i) Log Based Recovery :-

In log based recovery system, a log is maintained in which all the modifications of the DB are kept. A log consists of log records.

An update log record describes a single DB write and contains following fields.

a) Transaction Identifier :-
unique number is given to each transaction that performs write operation.

b) Data-Item Identifier :-
Unique number is given to data item written.

c) Old-value :- gives value of data-item before write.

d) New-value :- gives value of data item after write.

In log based recovery two operations are used :-

a) Redo :- It means work of the transactions that are completed successfully before crash is to be performed again. (Data items will take new values).

b) Undo :- It means all the work done by the transactions that did not complete due to crash is to be undone. (data items will take old values).

Scanned by CamScanner

(*) Two Types of log based recovery :-

1) Deferred Database Modification :-

In deferred database modification technique, all the write operations of any transaction Ti are deferred until it partially commits. It means modify DB after Ti partially commits.

In this technique old value field is not needed.

eg:- Transaction T1 transfers 200 Rs. from Account A to B and deposits 200 Rs in C.

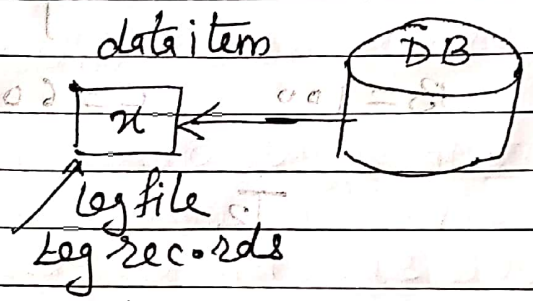
A = 500 B = 100 C = 600 initialy.

T1	T2
read(A);	
A := A − 200;	
write(A);	
read(B);	
B := B + 200;	read(C);
write(B);	C := C + 200;
	write(C);

Scanned by CamScanner

Corresponding log record is given as,

<T1, start>
<T1, A>
<T1, A, 300>
<T1, B>
<T1, B, 1200>
<T1, commit>
<T2, start>
<T2, C>
<T2, C, 800>
<T2, Commit>



System crash occurs

Commit found → redo

commit not found → delete (rejected)

a) If system crash occurs after write(A) of T1:-

<T1, start>
<T1, A>
<T1, A, 300>

No need to perform redo operation

Scanned by CamScanner

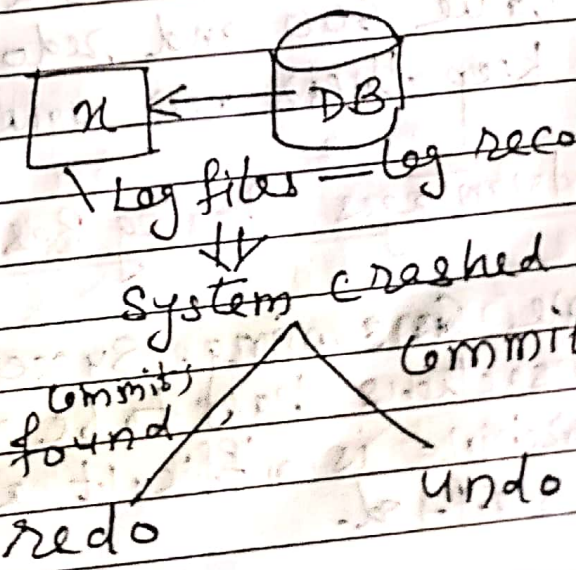
as no commit record appears and transaction T1 is rejected.

b) After write(c) of T_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>

In this case, you have to redo T_1 because both <T_1, start> and <T_1, commit> appears in log.
∴ after redo, A = 300, B = 1200

2) Immediate Database Modification :-



Scanned by CamScanner

In this both old as well as new values are needed.

<T1, Start>
<T1, A>
<T1, A, 500, 300>
<T1, B>
<T1, B, 1000, 1200>
<T1, Commit>
<T2, start>
<T2, C>
<T2, C, 600, 800>
<T2, Commit>

a) If system crash occurs after write(A) of T1.

Here only start exists so undo transaction T1. As a result account A restores its old value 500.

b) After write (C) of T2 :-

Undo T2 as a result C will have old value 600 and redo T1. A and B will keep their new value.

- **Disadvantages of log Based Recovery:-**

1) Time Consuming. Successfully completed transactions have to be redone.
2) Searching is difficult as whole log has to be scanned.

Scanned by CamScanner

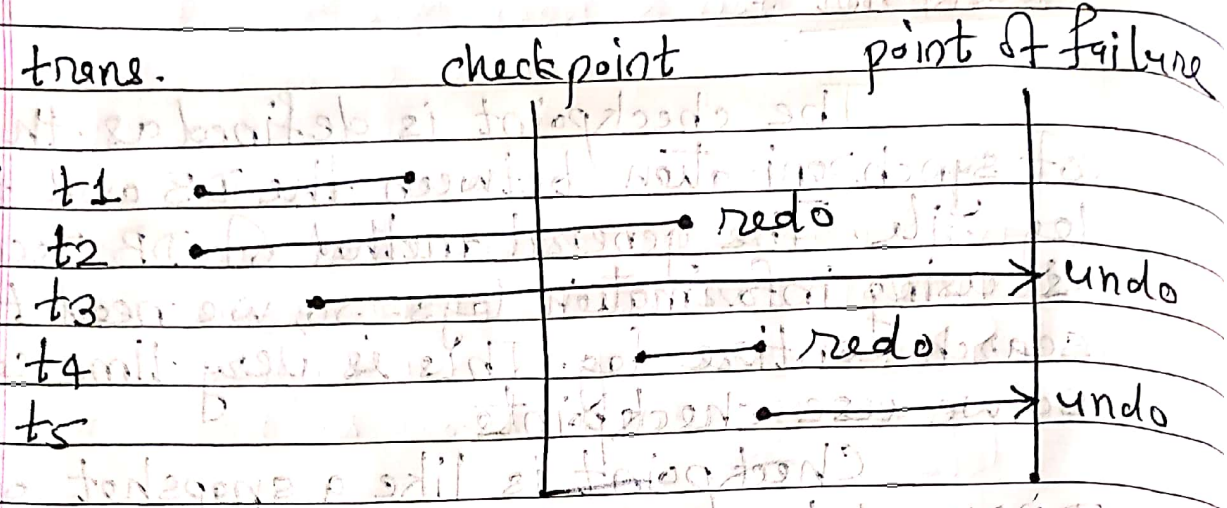
ii) Checkpoint :-

- The checkpoint is defined as the point of synchronization between the DB and trans. log file. The general method of DB recovery is using information logs. So, we need to search entire log. This is very time consuming so, we use checkpoints.
- Checkpoint is like a snapshot of the DBMS state. By taking snapshot periodically, DBMS can reduce the amount of work to be done ~~using~~ during restart in the event of system crash. The approach is to find a point i.e. sufficiently ~~far~~ far back to ensure that any item written before that point has been done correctly and saved safely. This method is called checkpointing.

<checkpoint> record gives a list of all transactions that were in progress at time the checkpoint was taken. Thus checkpoint help the system to provide the information at the restart time, which transactions to undo and which to redo.

eg:- Say transaction T commits prior to checkpoint. It implies that <T, commit> must appear in the log before the <checkpoint> record. Any DB modifications made by T must have been written to the disk either prior to the checkpoint or as a part of checkpoint itself. Thus at recovery time there is no need to perform redo operation on T.

Scanned by CamScanner



We observe that T1 completed successfully prior to the time tc, t2 started prior to tc but before time tf, T3 also started prior to tc but could not complete by time, tf, T4 started after time tc and complete successfully before time tf, also T5 transaction started after time tc but did not complete by time tf.

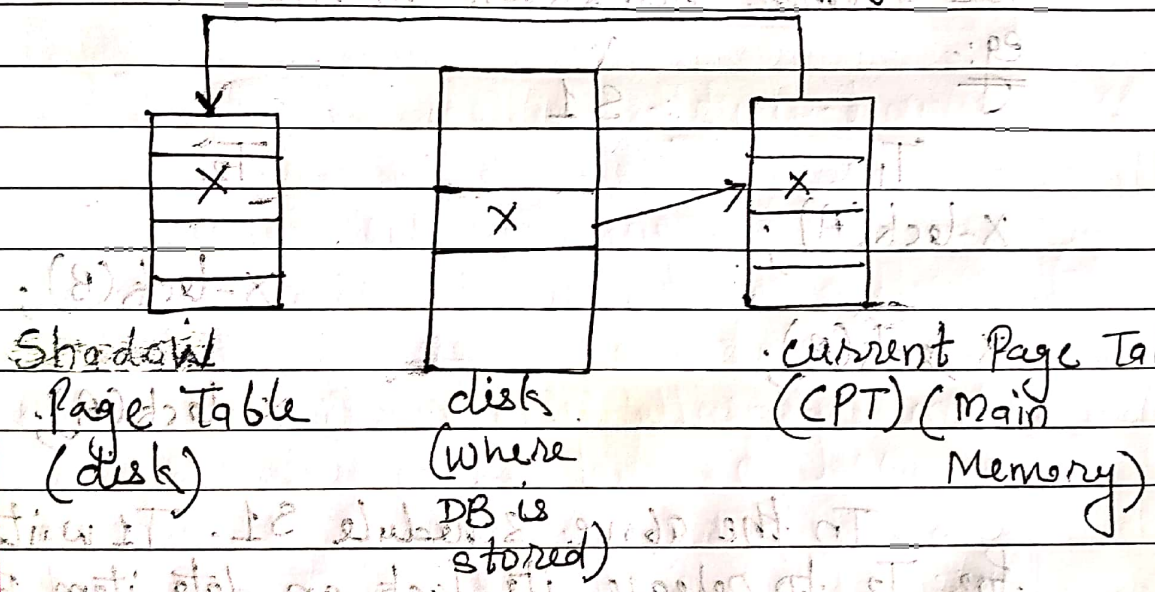
<u>Note</u> :- Checkpoint can be taken at a fixed interval of time, say every 15 minutes or 30 minutes or 1 hr and so on.

- Any transaction committed before checkpoint is successfully updated in DB.
- <To, start> before checkpoint <To, commit> after checkpoint ⇒ Redo.
- <To, start>, <to, commit>, after checkpoint ⇒ Redo.
- <To, start> before/after, no commit ⇒ undo
-

Shadow Paging :-

Shadow paging is an alternative technique for recovery technique to overcome the advantages of log based technique. The main idea of shadow paging is to keep two page tables in DB. One is used for current operations and other is used in case of recovery.

1) Shadow Page table :- This table cannot be changed during any transaction.
2) Current Page Table :- This page table may be changed during transaction.



If we want to perform operation on data item X in the DB stored on disk. It gets transferred from disk to main memory (CPT). CPT copies it to the shadow page table and operations are performed in CPT. If operation is successfull then new c/p is again copied in SPT and SPT transfers it in the place where DB is

stored. If operation is not successful then we go to shadow page table for old values.

Note: When we are performing operations on data item X initially both the tables (CPT, SPT) are identical. Changes are only done on CPT and SPT is used for recovery purpose.

* Deadlock :-

A system is in deadlock state if there exists a set of transactions such that for every transaction in the set is waiting for another transaction in the set.

eg:

S1

T_1	T_2
X-lock(A);	
	X-lock(B);
X-lock(B);	
	X-lock(A);

In the above schedule S1. T1 waits for T2 to release its lock on data-item B and T2 waits for T1 to release its lock. Such a cycle where transactions waiting for locks to be released is called as deadlock.

Scanned by CamScanner

(*) Methods for dealing with deadlock problem:-

1) Prevention of deadlock.
2) Detection of deadlock.

1) Deadlock Prevention :-

We can prevent deadlock by giving each transactions a priority and ensuring that the lower priority transactions are not allowed to wait for higher priority transactions or vice versa.

One way of assigning priorities is to give each transaction a timestamp when it starts.

Techniques used :-

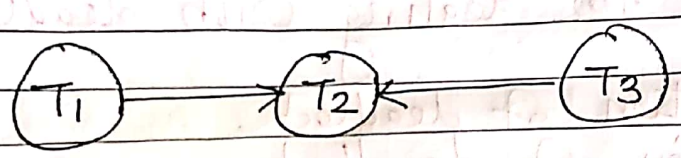
i) Wait-die
ii) Wound-wait.

1) Wait-die :-

If t_i has higher priority, it is allowed to wait else it is aborted. It mean that when T_i requests for a data item i.e. currently held by T_j, then T_i is allowed to wait only when it has a timestamp smaller than that of T_j i.e. T_i is older than T_j, otherwise T_i is rolled back (dies).

eg:- Say T_1 requests for a data item held by T_2 and T_3 also requests for data item held by T_2 i.e.

Scanned by CamScanner



T_1 requests T_2
T_3 also requests T_2

Then T_1 is made to wait while T_3 is rolled back provided $Ts(T_1)=3$, $Ts(T_2)=6$, $Ts(T_3)=9$. This scheme is non-preemptive

if $Ts(T_i) < Ts(T_j)$
→ T_i waits
else
→ T_i is rolled back.

2) <u>Wound-wait</u> :-

If transaction has higher priority then we abort T_j else T_i waits. It means ~~wait~~ that when T_i requests a data item currently held by T_j, then T_i is allowed to wait only if it has $Ts(T_i) > Ts(T_j)$. Else T_j is rolled back.

eg: If T_1 requests a data item held by T_2 then the data item will be preempted from T_2 & T_2 will be rolled back.

$T_1 \longrightarrow T_2$ (rollback)

Now if T_3 requests a data item held by T_2, then T_3 waits i.e.

$T_2 \longleftarrow T_3$ (waits)

This method is preemptive one.

Scanned by CamScanner

Note:- In wait-die scheme lower priority transactions can never wait for higher transactions whereas in wound-wait scheme the higher priority transactions never wait for lower priority transactions. In either case given above, no deadlock cycle can exist.

* Deadlock Detection :-

To detect deadlock wait for graph (WFG) is created.

WFG:- It consists a pair $G=(V,E)$ where V is set of vertices and E is the set of edges. Vertices show transactions and edges show dependency of transaction.

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 a graph. When T_i has obtained the required item from T_j remove the edge. Deadlock occurs when there is cycle in WFG.

Consider WFG as shown in fig.

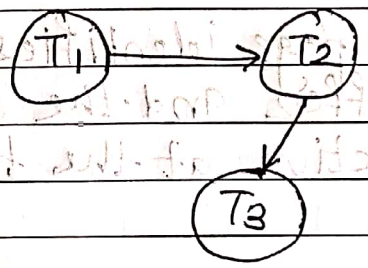
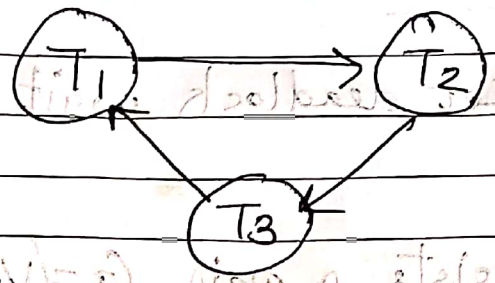


fig. WFG without deadlock situation.

There are 3 transactions T_1, T_2, T_3. The transaction T_1 is waiting for any data item held by T_2 & 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 (WFG).

Consider the graph as shown in fig. Where T_3 is also waiting for any data item held by T_1.



Thus, there exists a cycle in WFG.

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

Which results system in deadlock state & T_1, T_2, T_3 are all blocked.

* ARIES :- (Algorithm for Recovery & Isolation Exploiting Semantics) :-

It works in three phases:-

1) **Analysis** :-

This page identifies the dirty pages in the buffer and the set of transactions active at the time of system crash.

2) **Redo** :-

The redo phase actually reapplies updates from the log to the DB. It is applied only for committed transactions.

Scanned by CamScanner

3) Undo:-

during the undo phase the log is scanned backwoord and operations of transactions that were ~~waiting~~ active at the time of crash are undone in reverse order.

Scanned by CamScanner