Chapter 7
Transactions processing and Concurrency Control.

7.1 ACID properties.
7.2 Concurrent Executions.
7.3 Serializability Concept
7.4 Lock based protocols
7.5 Deadlock handling & prevention

Transaction concept

↳ An action, or series of actions, carried out by a single user or application program, which reads or updates the contents of the database is called transaction.

↳ A transaction is a logical unit of work on the database.

↳ During transaction execution the database may be inconsistent

↳ When the transaction is committed, the database must be consistent.

↳ Two main issues to deal with.

i. failures of various kinds, such as hardware failures and system crashes

ii. Concurrent execution of multiple transactions.

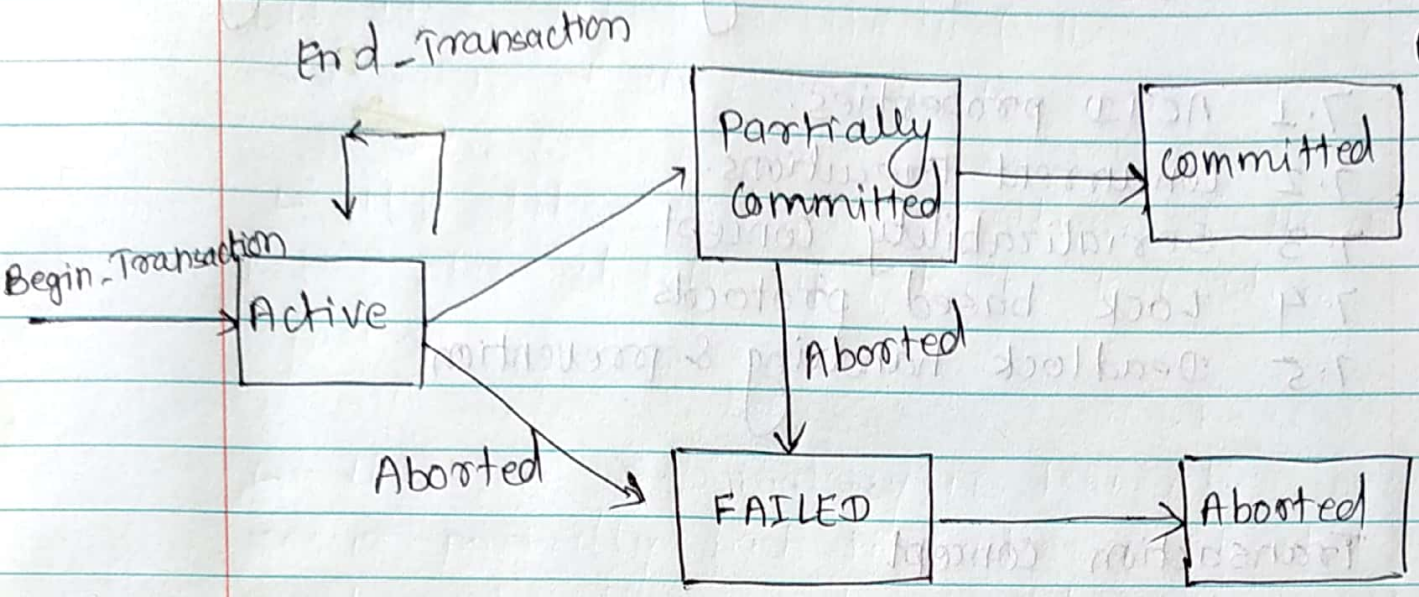


Figure: State transition diagram for a transaction.

Transaction state.

Active:

The initial state; the transaction stays in this state while it is executing.

Partially committed.

When a transaction executes its final opesation, it is said to be in a partially committed state

Failed:

Which occurs if the transaction can't be committed or transaction is aborted while in the Active state.

Aborted:

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

i. Re-start the transaction
ii. Kill the transaction.

Committed:

If a transaction executes all its opesations successfully, it is said to be committed. All its effects are now permanently established on the database system.

Properties of Transactions.

To preserve integrity of data, the database system must ensure four important properties of transactions called ACID properties.

Atomicity.

↳ The "all or nothing" property.
↳ A transaction is an invisible unit that is either performed in its entirety or is not performed at all.

↳ It is the responsibility of the recovery subsystem of the DBMS to ensure atomicity.

↳ It states that all operations of the transaction take place at once if not, the transaction is aborted.

↳ There is no midway, i.e. transaction cannot occur partially. Each transaction is treated as one unit and either run to completion or is not executed at all.

↳ Atomicity involves the following two opesations:

i. Abort:

If transaction aborts then all the changes made are not visible.

ii. Commit:

If a transaction commits then all the changes made are visible.

Example:

Lets assume that following transaction T consisting of T_1 and T_2. A consists of Rs 600 and B consists of Rs 300. Transfes Rs 100 from account A to account B.

T_1	T_2
Read (A)	Read (B)
A: A-100	B: B+100
write (A)	write (B)

After the completion of transaction, A consists of Rs 500 and B consists of Rs 400.

If the transaction T fails after the completion of transaction T_1 but before completion of T_2, then the amount will be deducted from A but not added to B. This shows the inconsistent database state. In order to ensure correctness of database state, the transaction must be executed entirety.

Consistency:

↳ The integrity constraints are maintained so that the database is consistent before and after the transaction.

↳ The execution of transaction will leave a database in either its prior stable state or a new stable state.

↳ The consistent property of the database states that every transaction sees a consistent database instance.

↳ The transaction is used to transform the database from one consistent state to another consistent state.

For example: The total amount must be maintained

before or after the transaction.

Total before T occurs = 600+300 = 900
Total after T occurs = 500+400 = 900

Therefore, the database is consistent. In the case when T_1 is completed but T_2 fails, then inconsistency will occur.

Isolation:

↳ It shows that the data which is used at the time of execution of a transaction cannot be used by the second transaction until the first one is completed.

↳ In isolation, if the transaction T_1 is being executed and using the data item X, then the data item can't be accessed by any other transaction T_2 until the transaction T_1 ends.

↳ The concurrency control subsystem of the DBMS enforced the isolation property.

Durability:

↳ The durability property is used to indicate the performance of the database's consistent state. It states that the transaction made the permanent changes.

↳ The effects of a sucessfully completed (committed) transaction are permanently recorded in the database and mustnot be lost because of a subsequent failure.

↳ They cannot be lost by the erroneous operation of a faulty transaction or by the system failure. When a transaction is completed, then the database reaches a state known as the consistent state. That consisted state cannot be lost, even in the event of a system's failure.

↳ The recovery subsystem of the DBMS has the responsibility of Dusabilcy property.

Concurrent Eaecution.

↳ Multiple transactions are allowed to run concurrently in the system.

↳ The process of managing simultaneous operations on the database without having them interfese with one another is called concurrency control.

↳ The operations of transactions are interleaved to achieve conurrent eaecution.

Advantages are:

Increased processor and disk utilization, leading to better transaction throughput: one transaction can be using the CPU while another is reading from or writing to the disk.

Reduced average response time for transactions; short transactions need not wait behind long ones.

Concurrent control schemas.

Mechanisms to achieve isolation, i.e., to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database.

Schedule:

↳ A sequence of the operations by a set of concurrent transactions that preserves the order of the operations in each for the individual transactions.

↳ A schedule for a set of transactions must consist of all instructions of those transactions.

↳ Must preserve the order in which the instructions appear in each individual transaction.

↳ A transaction that successfully completes its execution will have a commit instructions as the last statement.

↳ A transaction that fails to successfully

complete its execution will have an abort instructions as the last statement.

↳ A serial schedule in which T_1 is followed by T_2:

Schedule 1:

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)

↳ A serial schedule where T_2 is followed by T_1 : schedule 2

T_1	T_2
	read(A)

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

Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is equivalent to schedule 1. In schedules 1, 2 and 3, the sum A + B is preserved.

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

write (B)	
	read (B)
	B := B + temp
	write (B)

The following concurrent schedule doesnot preserve the value of (A+B).

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

Serial schedule.

↳ A schedule where the operations of each transaction are executed consecutively without any interleaved operations form other transactions

↳ There is no interference between transactions.

Nonserial Schedule.

A schedule where the operations form a set of concurrent transactions are interleaved.

Serializability

↳ Serializability is concurrency scheme where the concurrent transaction is equivalent to one that executes the transactions serially.

↳ A schedule is list of transactions.

↳ A schedule S of n transactions is serializable if it is equivalent to some serial schedule of the same n transactions.

↳ Two schedules are called result equivalent if they produce the same final state of the database.

↳ A schedule S is serial if, for every transaction T participating in the schedule, all the operations of T is executed consecutively in the schedule.

↳ A schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to notions of :

i. conflict serializability

ii. View serializability

↳ The main objective of serializability is to find non-serial schedules that allow transactions to execute concurrently without interfesene and produce a database state that could be produced by a serial execution.

Conflict Serializability.

↳ Two actions Ai and Aj executed on the same data object by Ti and Tj conflicts if either one of them is write operation.

↳ Conflict serializability defines two instructions of two different transactions accessing the same data item to perform a read/write operation.

↳ Let Ai and Aj are consecutive non-conflicting actions that belong to different transactions. We can swap Ai and Aj without changing the result.

↳ If two transactions are both read operation, then they are not in conflict.

↳ If one transaction wants to perform a read operation and other transaction wants to perform a write operation, then they are in conflict and cannot be swapped.

↳ If both the transactions are for write operation, then they are in conflict, but can be allowed to take place in any order, because the transactions donot read the value updated by each other.

↳ If a schedule S can be transformed into schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are conflict equivalent.

↳ We can say that a schedule S is conflict serializable if it is conflict equivalent to a serial schedule.

↳ Actions I_i and I_j of transactions T_i & T_j respectively, conflict if and only if there exists some item Q accessed by both I_i and I_j, and at least one of these instructions wrote Q.

I_i = read (Q), I_j = read(Q) I_i & I_j don't conflict

I_i = read(Q), I_j = write(Q) They conflict.

I_i = write(Q), I_j = read(Q) They conflict

I_i = write(Q), I_j = write(Q) They conflict

↳ Intuitively, a conflict between I_i and I_j forces a (logical) temporal order between them. That is, replacing their order will change the result.

↳ If I_i and I_j are consecutive in a schedule and they donot conflict, their results would remain the same even if they had been interchanged in the schedule.

Example:

Schedule 3 can be transformed into schedule 4, a serial schedule where T_2 follows T_1, by series of swaps of non conflicting instructions

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

Schedule 3

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

Schedule 4

Schedule 3 is conflict serializable

↳ Example of a schedule that is not conflict serializable. We are unable to swap instructions

in the above schedule to obtain either the serial schedule $\langle T_3, T_4 \rangle$ or the serial schedule $\langle T_4, T_3 \rangle$

View Sesializability

A schedule will be view serializable if it is view equivalent to a sesial schedule.

If a schedule is conflict sesializable, then it will be view serializable.

The view serializable which doesnot conflict serializable contains blind writes.

View Equivalent.

Two schedules S_1 and S_2 are said to be view ~~serializable~~ equivalent if they satisfy the following conditions.

1. Initial Read.

An initial read of both schedules must be the same. Suppose two schedule S_1 and S_2. In schedule S_1, if a transaction T_1 is reading the data item A, then in S_2, transaction T_1 should also read A.

T_1	T_2
Read(A)	
	Write(A)

schedule S_1

T_1	T_2
	write(A)
Read(A)	

schedule S_2

Above two schedules are view equivalent because initial read operation in S_1 is done by T_1 and in S_2 it is also done by T_1.

2. Updated Read:

In schedule S_1, if T_i is reading A which is updated by T_j then in S_2 also, T_i should read A which is updated by T_j

T_1	T_2	T_3
write(A)		
	write(A)	
		Read(A)

Schedule S_1

T_1	T_2	T_3
	write(A)	
write(A)		
		Read(A)

Schedule S_2

Above two schedules are not view equal because in S_1 T_3 is reading A updated by T_2 In S_2, T_3 is reading A updated by T_1.

3. Final write:

A final write must be the same between both the schedules. In schedule S_1, if a transaction T_1 updates A at last then in S_2, final write operation should also be done by T_1

T_1	T_2	T_3
write(A)		
	Read(A)	
		write(A)

Schedule S_1

T_1	T_2	T_3
	Read(A)	
write(A)		
		write(A)

Schedule S_2

Above two schedule is view equal because final write operation in S_1 is done by T_3 and in S_2, the final write operation is also done by T_3

Example:

T_1	T_2	T_3
Read(A)		
	write(A)	
write(A)		
		write(A)

Schedule S

With 3 transactions, the total number of possible schedule.

$= 3! = 6$

$S_1 = \langle T_1\ T_2\ T_3 \rangle$
$S_2 = \langle T_1\ T_3\ T_2 \rangle$
$S_3 = \langle T_2, T_3, T_1 \rangle$
$S_4 = \langle T_2\ T_1\ T_3 \rangle$
$S_5 = \langle T_3\ T_1\ T_2 \rangle$
$S_6 = \langle T_3\ T_2\ T_1 \rangle$

Taking first schedule S_1:

T_1	T_2	T_3
Read (A)		
write(A)		
	write(A)	
		write(A)

Schedule S_1

Step 1: final updation of data items.

In both schedules S and S_1, there is no read except the initial read that's why we don't need to check that condition.

Step 2: Initial Read:

The initial read operation in S is done by T_1 and in S_1 it is also done by T_1

step 3: Final write:

The final write operation in S is done by T_3 in S_1, it is also done by T_3 in S. So, S & S_1 are view equivalent.

The first schedule S_1 satisfies all three conditions so we don't need to check another schedule.

Hence, view equivalent serial schedule is

$T_1 \rightarrow T_2 \rightarrow T_3$

Difference between conflict and view serializability.

- Conflict serializability is easy to achieve but view serializability is difficult to achieve.
- Every conflict serializable but the reverse is not true.
- It is easy to test conflict serializability but expensive to test view serializability.
- Most of the concurrency control schemas used in practice are based on conflict serializability.

Lock-based Protocol.

In this type of protocol, any transaction cannot read or write data until it acquires an appropriate lock on it. There are two types of lock:

i. Shared lock. (S)

↳ It is also known as a Read-only lock.

↳ In shared lock, the data item can only read by the transaction.

↳ It can be shared between the transactions because when the transaction holds a lock, then it can't update the data on the data item.

↳ If a transaction T_i has obtained a sharedlock on item Q, then T_i can read but can't write Q.

ii. Exclusive lock: (X)

↳ In the exclusive lock, the data item can be both read as well as written by the transaction.

↳ This lock is exclusive, and in this lock, multiple transactions donot modify the same data simultaneously.

↳ If a transaction T_i has obtained exclusive lock on item Q, then T_i can both read and write Q

Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

Lock-compatibility matrix:

		current state of locking of data items		
		Unlocked	Shared(S)	Exclusive(X)
Lock mode of request	unlocked		Yes	Yes
	Shared(S)	Yes	Yes	NO
	Exclusive(X)	Yes	NO	NO

↳ A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions.

↳ Any number of transactions can hold shared locks on an item, but if any transaction holds an exclusive lock on the item no other transaction may hold any lock on the item.

↳ If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been

released. The lock is then granted.

Pitfalls of Lock-Based protocols.

Consider the partial schedule.

T_3	T_4
lock-X(B)	
read(B)	
B := B-50	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

↳ Neither T_3 nor T_4 can make progress - executing lock-S(B) causes T_4 to wait for T_3 to release its lock on B, while executing lock-X(A) causes T_3 to wait T_4 to release its lock on A.

↳ Such situation is called a deadlock.

↳ To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.

↳ The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.

↳ Starvation is also possible if concurrency control manager is badly designed. For example:

A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.

The same transaction is repeatedly rolled back due to deadlocks.

↳ Concurrency control manager can be designed to prevent starvation.

The two-phase Locking Protocol.

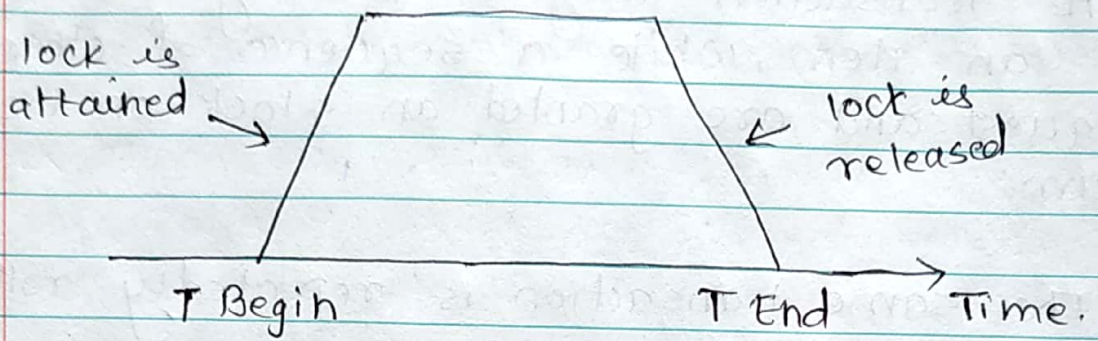
↳ This is a protocol which ensures conflict-serializable schedules.

↳ The two phase locking protocol divides the execution phase of the transaction into three parts.

↳ In the first part, when the execution of the transaction starts, it seeks permission for the lock it requires.

↳ In the second part, the transaction acquires all the locks. The third phase is started as soon as the transaction releases its first lock.

↳ In the third phase, the transaction cannot demand any new locks. It only releases the acquired locks.



There are two phases of 2PL:

Growing phase:

In the growing phase, a new lock on the data item may be acquired by the transaction, but none can be released.

Shrinking Phase:

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

In the below example, if lock conversion is allowed then the following phase can happen:

1. Upgrading of lock (from S(a) to X(a)) is allowed in growing phase.

2. Downgrading of lock (from X(a) to S(a)) must

be done in shrinking phase.

Example:

	T_1	T_2
0	LOCK-S(A)	
1		LOCK-S(A)
2	LOCK-X(B)	
3	—	—
4	UNBLOCK(A)	
5		LOCK-X(C)
6	UNBLOCK(B)	
7		UNBLOCK(A)
8		UNBLOCK(C)
9	—	—

The following way shows how unlocking and locking work with 2-PL

Transaction T_1:
- Growing phase: from step 0-2
- Shrinking phase: from step 4-6
- Lock-point : at 3

Transaction T_2:
- Growing phase: from step 1-5
- Shrinking phase: from step 7-8
- Lock point : at 6

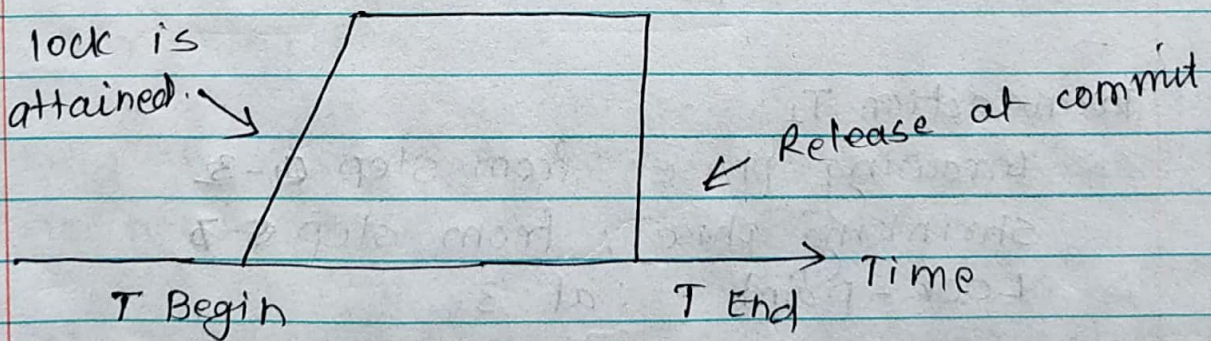
Strict Two-phase locking (strict-2PL)

↳ The first phase of strict-2PL is similar to 2PL. In the first phase, after acquiring all the locks, the transaction continues to execute normally.

↳ The only difference between 2PL and strict-2PL is that strict-2PL doesnot release a lock after using it.

↳ strict-2PL waits until the whole transaction to commit, and then releases all the locks at a time.

↳ strict 2PL protocol doesnot have shrinking phase of lock release.



↳ It doesn't have cascading abort as 2PL does.

Lock Conversions.

Two phase locking with lock conversions.

First phase:
- can acquire a lock-S on item.
- can acquire a lock-X on item.
- can convert a lock-S to a lock-X (upgrade)

second phase:
- can release a lock-S
- can release a lock-X
- can convert a lock-X to lock-S (downgrade)

This protocol assures serializability. But still relies on the programmer to insert the various locking instructions.

Deadlock:

↳ A deadlock is a condition where two or more transactions are waiting indefinitely for one another to give up locks.

↳ Deadlock is said to be one of the most feared complications in DBMS as no task ever gets finished and is in waiting state forever.

For example:

In the student table, transaction T_1 holds a

lock on some rows and needs to update some rows in the grade table. Simultaneously, transaction T_2 holds locks on some rows in the grade table and needs to update the rows in the student table held by Transaction T_1.

Now, transaction T_1 is waiting for T_2 to release its lock and similarly, transaction T_2 is waiting for T_1 to release its lock. All activities come to halt state and remain at a standstill. It will remain in a standstill until the DBMS detects the deadlock and aborts one of the transactions.

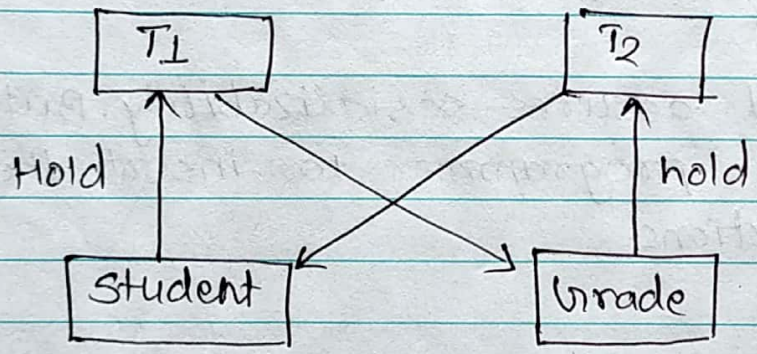


Figure: Deadlock in DBMS

↳ In a multi-process system, deadlock is an unwanted situation that arises in a shared resource environment, where a process indefinitely waits for a resource that is held by another process.

↳ System is deadlocked if there is a set of transactions such that every transaction in the

set is waiting for another transaction in the set.

Consider the following two transactions:

T_1: write (X)
write (Y)

T_2: write (Y)
write (X)

Schedule with deadlock:

T_1	T_2
lock-X on X	
write (X)	
	lock-X on Y
	write (Y)
	wait for lock-X on X
	write (X)
wait for lock-X on Y	
write (Y)	

↳ To prevent any deadlock situation in the system, the DBMS aggressively inspects all the operations, whose transactions are about to execute.

↳ The DBMS inspects the operations and analyzes if they can create a deadlock situation. If it finds that a deadlock situation might occur, then that transaction is never allowed to be executed.

↳ Deadlock prevention protocols ensure that the

system will never enter into a deadlock state.

Deadlock Avoidance.

↳ When a database is stuck in a deadlock state, then it is better to avoid the database rather than aborting or restasting the database. This is a waste of time and resource.

↳ Deadlock avoidance mechanism is used to detect any deadlock situation in advance. A method like "wait for graph" is used for detecting the deadlock situation but this method is suitable only for the smaller database. For the larger database, deadlock prevention method can be used.

Deadlock Detection.

In a database, when a transaction waits indefinitely to obtain a lock, then the DBMS should detect whether the transaction is involved in a deadlock or not. The lock manager maintains a wait for graph to detect the deadlock cycle in the database.

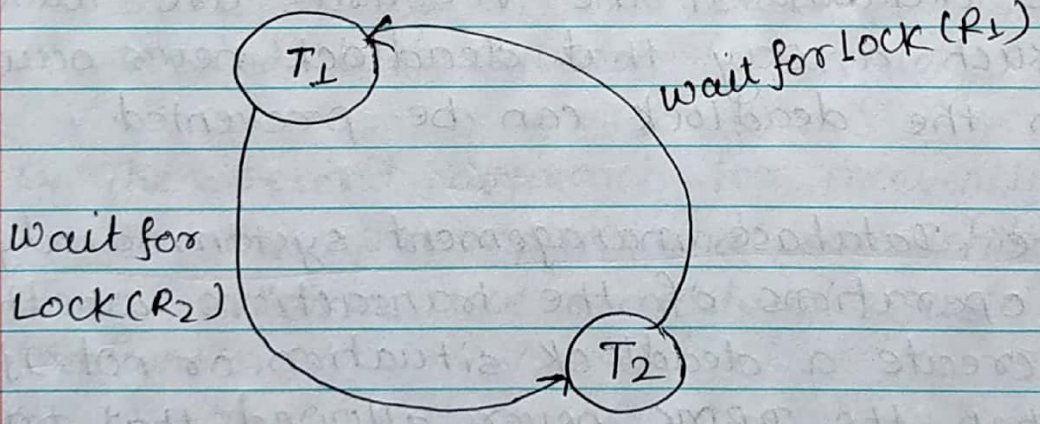
Wait for Graph

This is the suitable method for deadlock

detection. In this method, a graph is created based on the transaction and their lock. If the created graph has a cycle or closed loop, then there is a deadlock.

↳ The wait for the graph is maintained by the system for every transaction which is waiting for some data held by the others. The system keeps checking the graph if there is any cycle in the graph.

↳ The wait for a graph for the above scenario is shown below:



↳ Deadlocks can be described as a wait-for graph, which consists of a pair $G = (V, E)$.

V is a set of vertices (all the transactions in the system)

E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$

↳ If $T_i \rightarrow T_j$ is in E, then there is a directed edge from T_i to T_j, implying that T_i is waiting for T_j to release a data item.

↳ When T_i requests a data item currently being held by T_j, then the edge $T_i T_j$ is insested in the wait-for graph. This edge is removed only when T_j is no longer holding a data item needed by T_i.

Deadlock Prevention.

↳ Deadlock prevention method is suitable for a large database. If the resources are allocated in such a way that deadlock never occurs, then the deadlock can be prevented.

↳ The Database management system analyzes the operations of the transactions whethes they can create a deadlock situation or not. If they do, then the DBMS never allowed that transaction to be executed.

↳ Two approach to deadlock prevention:

One approach ensures that no cyclic waits can occur by ordering the requests for locks, requiring all locks to be acquired together.

Another approach performs transaction rolled

back instead of waiting for a lock, whenever the wait could potentially result in a deadlock.

↳ The first approach requires that each transaction locks all its data items before execution. Moreover, either all are locked in one step or none are locked. There are two main disadvantages.

It is often hard to predict, before the transaction begins, what data items need to be locked.

Data items utilization may be very slow, since many of the data items may be locked but unused for a long time.

↳ The second approach for preventing deadlocks is to use preemption and transaction rollback. These schemes use timestamps just for deadlock prevention.

Wait - Die Scheme.

↳ In this scheme, if a transaction requests for a resource which is already held with a conflicting lock by another transaction then the DBMS simply checks the timestamp of both transactions. It allows older transaction to wait until the resource is available for execution.

↳ When transaction T_i requests a data item

currently held by T_j, T_i is allowed to wait only if it has a timestamp smaller than that of T_j. Otherwise, T_i rolled back (dies).

Lets assume there are two transactions T_i and T_j and let TS(T) is a timestamp of any transaction T. If T_2 holds a lock by some other transaction and T_1 is requesting for resources held by T_2 then the following actions are performed by DBMS.

i. Check if $TS(T_i) < TS(j)$ - If T_i is older transaction and T_j has held some resource, then T_i is allowed to wait until the data item is available for execution. This means if the older transaction is waiting for a resource which is locked by the younges transaction, then older transaction is allowed to wait for resource until it is available.

ii. Check if $TS(T_i) < TS(T_j)$ - If T_i is older transaction and has held some resource and if T_j is waiting for it, then T_j is killed and restarted later with the random delay but with the same timestamp.

Example 2:

Suppose that transaction T_1, T_2 & T_3 have timestamps 20, 30 and 40 respectively. If T_1 request a data item

held by T_2, then T_1 will wait. If T_3 requests a data item held by T_2, then T_3 will be rolled back.

Wound wait scheme:

↳ In wound wait scheme, if the older transaction requests for a resource which is held by the younger transaction, then older transaction forces younger one to kill the transaction and release the resource. After the minute delay, the younger transaction is restarted but with the same timestamp.

↳ If the older transaction has held a resource which is requested by the younger transaction, then the younger transaction is asked to wait until older release it.

↳ When transaction T_i requests a data item currently held by T_j, T_j is allowed to wait only if it has a timestamp larger than that of T_j. Otherwise, T_i rolled back.

↳ For example: suppose that transaction T_1, T_2 & T_3 have timestamps 20, 30 and 40 respectively. If T_1 request a data item held by T_2, then the data item will be preempted from T_2 & T_2 will be rolled back. If T_3 requests a data item held by T_2, then T_3 will wait.

↳ Both in wait die and in wound-wait schemas, a rolled back transaction is restasted with its original timestamp. Older transactions thus have precedence over newer ones & stasvation is hence avoided.

Timeout-Based Schemas:

↳ A transaction waits for a lock only for a specified amount of time. After that the wait times out and transaction is rolled back. Thus deadlocks are not possible.

↳ simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.