CONCURRENCY CONTROL

- Now we understood that if there is a schedule how to check whether it will work correctly or not i.e. weather it will maintain the consistency of the data base or not. (conflict serializability, view serializability, recoverability and cascade less)

- Now we will understand those protocol which guarantee how to design those schedules which ensure conflict serializability or other properties.

- There are different approach or idea to ensure conflict serializability which is the most important property. So first we must understand what is the possibility of conflict between two instruction and if somehow, we manage than the generated schedule will always be conflict serializable

- If we remember, two instructions are conflicting if and only if three things happen simultaneously

    - Belong to different transactions
    - Must operate on same data value
    - At least one of them should be a write instruction \
- if we think sufficiently, there will be no way to change any of these three conditions. But actual problem is not that two instructions are trying to access same data base but, they are trying to do that at same time.
- So point if we somehow by any technique manage that two transaction do not access same data at same time then ensuring conflict serializability will be easy

- Now question is how to approach conflict serializability, there are two popular approaches to go forwards.

    - **Time stamping based method: -** where before entering the system, a specific order is decided among the transaction, so in case of a clash we can decide which one to allow and which to stop.
    - **Lock based method: -** where we ask a transaction to first lock a data item before using it. So that no different transaction can use a data at the same time, removing any possibility of conflict.
        - 2 phase locking
            - Basic 2pl
            - Conservative 2pl
            - Rigorous 2pl
            - Strict 2pl
        - Graph based protocol
    - **Validation based protocol –** Majority of transactions are read only transactions, the rate of conflicts among the transaction may be low, thus many of transaction, if executed without the supervision of a concurrency control scheme, would nerveless leave the system in a consistent state.

- **Goals of a Protocol**: - We desire the following properties from schedule generating protocols
    - Concurrency should be as high as possible, as this is our ultimate goal because of which we are making all the effort.
        - The time taken by a transaction should also be less.
    - Properties satisfied by the protocol
    - Easy to understand and implement
    - In the last after discussing all the protocols we will compare different protocol only on the bases of degree of concurrency they provide and along which how many prope
    - rties they ensure.

TIME STAMP ORDERING PROTOCOL

- Basic idea of time stamping is to decide the order between the transaction before they enter in the system using a stamp (time stamp), in case of any conflict during the execution order can be decided using the time stamp.
- Let's understand how this protocol works, here we have two idea of timestamping, one for the transaction, and other for the data item.
- Time stamp with transaction,
    - With each transaction t_i, in the system, we associate a unique fixed timestamp, denoted by $TS(t_i)$. this timestamp is assigned by database system to a transaction at time transaction enters into the system. If a transaction has been assigned a timestamp $TS(t_i)$ and a new transaction t_j , enters into the system with a timestamp $TS(t_j)$, then always $TS(t_i) < TS(t_j)$.
    - Two things are to be noted, first time stamp of a transaction remain fixed throughout the execution, second it is unique means no two transaction can have the same timestamp.
    - The reason why we called time stamp not stamp, because for stamping we use the value of the system clock as stamp, advantage is, it will always be unique as time never repeats and there is no requirement of refreshing and starting with fresh value.
    - The time stamp of the transaction also determines the serializability order. Thus if $TS(t_i) < TS(t_j)$, then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction t_i appears before transaction t_j.

- Time stamp with data item
    - In order to assure such scheme, the protocol maintains for each data item Q two timestamp values:
    - **W-timestamp(Q)** is the largest time-stamp of any transaction that executed write(Q) successfully.
    - **R-timestamp(Q)** is the largest time-stamp of any transaction that executed read(Q) successfully.
    - These timestamps are updated whenever a new read(Q) or write(Q) instruction is executed.
- Suppose a transaction Ti request a ***read(Q)***
    - If **TS(Ti) < W-timestamp(Q),** then Ti needs to read a value of Q that was already overwritten. Hence, the read operation is rejected, and Ti is rolled back.
    - If **TS(Ti)≥ W-timestamp(Q),** then the read operation is executed, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(T_i).
- Suppose that transaction Ti issues ***write(Q).***
    - If **TS(Ti) < R-timestamp(Q),** then the value of Q that Ti is producing was needed previously, and the system assumed that that value would never be produced. Hence, the write operation is rejected, and Ti is rolled back.
    - If **TS(Ti) < W-timestamp(Q),** then Ti is attempting to write an obsolete value of Q. Hence, this write operation is rejected, and Ti is rolled back.
    - If **TS(Ti) ≥ R-timestamp(Q),** then the write operation is executed, and W-timestamp(Q) is set to max(W-timestamp(Q), TS(Ti)).
    - If **TS(Ti) ≥ W-timestamp(Q),** then the write operation is executed, and W-timestamp(Q) is set to max(W-timestamp(Q), TS(Ti)).
- If a transaction T_i is rolled back by the concurrency control scheme as a result of either a read or write operation, the system assigns it's a new timestamp and restarts it.

Properties

Conflict serializability	View serializability	recoverability	Cascade less ness	Deadlock Independence
yes	yes	No	no	Yes

- Time stamp ordering protocol ensures conflict serializability. Because conflicting operations are processed in timestamp order, since all the arcs in the precedence graph are of the form thus, there will be no cycles in the precedence graph.
- As we know that view is liberal form conflict so view serializability also holds good
- As there is a possibility of dirty read, and no restriction on when to commit, so can be irrecoverable and may suffer from cascading rollback.
- At the time of request, here either we allow or we reject, so there is no idea of deadlock.
- If a schedule is not conflict serializable then it is not allowed by time stamp ordering scheme.
- But it is not necessary that all conflict serializable schedule generated by time stamping.
- Further modifications are possible if we want to ensure recoverability and cascade lessness, using different approaches
    - By performing all writes together at the end of the transaction, i.e. while writers are in progress, no transaction is permitted to access any of data items that have been written.
    - By using a limited form of locking, where by read of uncommitted items are postponed until the transaction that uploaded the item commit.
    - Recoverability alone can be ensured by tracking uncommitted writes and allowing a transaction t_i to commit only after the commit of any transaction that wrote a value that t_i read.

Conclusion: -

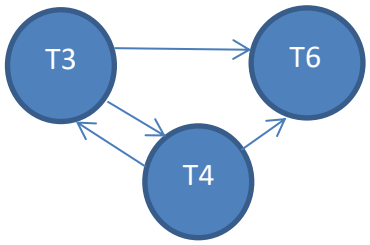
- It may cause starvation to occur, as if a sequence of conflicting short transactions causes repeated restarting of the long transaction, and then there is a possibility of starvation of long transaction.
- It is relatively slow as before executing every instruction we have to check conditions before. Time stamping protocol ensure that the schedule designed through this protocol will always be conflict serializable.
- This protocol can also be used for determining the serializability order (order in which transaction must execute) among the transaction in advance.

THOMAS WRITE RULE

- Thomas write is an improvement in time stamping protocol, which makes some modification and may generate those protocols that are even view serializable, because it allows greater potential concurrency.
- It is a Modified version of the timestamp-ordering protocol in which obsolete write operations may be ignored under certain circumstances.
- The protocol rules for read operations remain unchanged. while for write operation, there is slightly change in Thomas write rule than timestamp ordering protocol.
- **When Ti attempts to write data item Q, if TS(Ti) < W-timestamp(Q),** then Ti is attempting to write an obsolete value of {Q}. Rather than rolling back Ti as the timestamp ordering protocol would have done, this {write} operation can be ignored. Otherwise this protocol is the same as the timestamp ordering protocol.
- This modification is valid as the any transaction with **TS(Ti) < W-timestamp(Q),** the value written by this transaction will never be read by any other transaction performing Read(Q) ignoring such obsolete write operation is considerable.
- Thomas' Write Rule allows greater potential concurrency. Allows some view-serializable schedules that are not conflict serializable.
- ***E.g.-***

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

Precedence Graph



- Above schedule is not allowed in timestamp ordering as the schedule is not conflict serializable (precedence graph consist of cycle) , while the schedule is allowed in Thomas write rule protocol because it ignores the write(Q) operation of T3,being it an obsolete updation to a value.And equivalent serial schedule is so T3->T4->T6.

Q In a database system, unique timestamps are assigned to each transaction using Lamport's logical clock. Let TS(T_1) and TS(T_2) be the timestamps of transactions T_1 and T_2 respectively. Besides, T_1 holds a lock on the resource R and T_2 has requested a conflicting lock on the same resource R. The following algorithm is used to prevent deadlocks in the database system assuming that a killed transaction is restarted with the same timestamp.

if TS(T_2)<TS(T_1) then
    T_1 is killed
else T_2 waits.

Assume any transaction that is not killed terminates eventually. Which of the following is TRUE about the database system that uses the above algorithm to prevent deadlocks?
(GATE-2017) (2 Marks)
(a) The database system is both deadlock-free and starvation-free
(b) The database system is deadlock-free, but not starvation-free
(c) The database system is starvation-free, but not deadlock-free
(d) The database system is neither deadlock-free nor starvation-free
Ans: a

Lock Based Protocols

- To ensure isolation is to require that data items be accessed in a mutually exclusive manner i.e. while one transaction is accessing a data item, no other transaction can modify that data item. Locking is the most fundamental approach to ensure this. Lock based protocols ensure this requirement. Idea is first obtaining a lock on the desired data item then if lock is granted then perform the operation and then unlock it.
- In general, we support two modes of lock because, to provide better concurrency.
- A data item can be locked in two modes-
- ***Shared mode***
    - If transaction T_i has obtained a shared-mode lock (denoted by S) on any data item Q, then T_i can read, but cannot write Q, any other transaction can also acquire a shared mode lock on the same data item(this is the reason we called this shared mode).
- **Exclusive mode**
    - If transaction T_i has obtained an exclusive-mode lock (denoted by X) on any data item Q, then T_i can both read and write Q, any other transaction cannot acquire either a shared or exclusive mode lock on the same data item. (this is the reason we called this exclusive mode)
- ***Lock –Compatibility Matrix***

Requested Lock \ Current State of lock of data items	Exclusive	Shared	Unlocked
Exclusive	N	N	Y
Shared	N	Y	Y
Unlock	Y	Y	-

- Conclusion shared is compatible only with shared while exclusive is not compatible either with shared or exclusive.
- To access a data item, transaction T_i must first lock that item, if the data item is already locked by another transaction in an incompatible mode, or some other transaction is already waiting in non-compatible mode, then concurrency control manager will not grant the lock until all incompatible locks held by other transactions have been released. The lock is then granted.
- ***Pitfalls of lock-based protocols-*** Lock based protocol *do not ensure serializability* as granting and releasing of lock do not follow any order and any transaction any time may go for lock and unlock. Here in the example below we can see, that even this transaction

in using locking but neither it is conflict serializable nor independent from deadlock. Even others problems also persist like non – recoverability or cascading rollbacks may occur.

S

T_1	T_2
LOCK-X(A)	
READ(A)	
WRITE(A)	
UNLOCK(A)	
	LOCK-S(B)
	READ(B)
	UNLOCK(B)
LOCK-X(B)	
READ(B)	
WRITE(B)	
UNLOCK(B)	
	LOCK-S(A)
	READ(A)
	UNLOCK(A)

- If we do not use locking, or if we unlock data items too soon after reading or writing them, we may get inconsistent states, as there exists a possibility of dirty read. On the other hand, if we do not unlock a data item before requesting a lock on another data item, deadlocks may occur and concurrency will be poor.
- We shall require that each transaction in the system follow a set of rules, called a **locking protocol**, indicating when a transaction may lock and unlock each of the data items for e.g. 2pl or graph based locking.
- Locking protocols restrict the number of possible schedules. The set of all such schedules is a proper subset of all possible serializable schedules.
- We say that a schedule S is **legal** under a given locking protocol if S is a possible schedule for a set of transactions that follows the rules of the locking protocol. We say that a locking protocol **ensures** conflict serializability if and only if all legal schedules are conflict serializable.
- When a transaction requests a lock on a data item in a particular mode, and no other transaction has a lock on the same data item in a conflicting mode, the lock can be granted. However, care must be taken to avoid the following scenario. Suppose a transaction T_2 has a shared-mode lock on a data item, and another transaction T_1 requests an exclusive-mode lock on the data item. Clearly, T_1 has to wait for T_2 to release the shared-mode lock. Meanwhile, a transaction T_3 may request a shared-mode lock on the same data item. The lock request is compatible with the lock granted to T_2, so T_3 may be granted the shared-mode lock. At this point T_2 may release the lock, but still T_1 has to wait for T_3 to finish. But again, there may be a new transaction T_4 that requests a shared-mode lock on the same data item, and is granted the lock before T_3 releases it.

In fact, it is possible that there is a sequence of transactions that each requests a shared-mode lock on the data item, and each transaction releases the lock a short while after it is granted, but T_1 never gets the exclusive-mode lock on the data item. The transaction T_1 may never make progress, and is said to be **starved**.

- We can avoid starvation of transactions by granting locks in the following manner: When a transaction T_i requests a lock on a data item Q in a particular mode M, the concurrency-control manager grants the lock provided that:
    - There is no other transaction holding a lock on Q in a mode that conflicts with M.
    - There is no other transaction that is waiting for a lock on Q and that made its lock request before T_i.
- a refinement of the basic two-phase locking protocol, in which **lock conversions** are allowed. We shall provide a mechanism for upgrading a shared lock to an exclusive lock, and downgrading an exclusive lock to a shared lock. We denote conversion from shared to exclusive modes by **upgrade**, and from exclusive to shared by **downgrade**. Lock conversion cannot be allowed arbitrarily. Rather, upgrading can take place in only the growing phase, whereas downgrading can take place in only the shrinking phase.

Two phase locking protocol(2PL)

- The protocol ensures that each transaction issue lock and unlock requests in two phases, note that each transaction will be 2 phased not schedule.
- Growing phase- A transaction may obtain locks, but not release any locks.
- Shrinking phase- A transaction may release locks, but may not obtain any new locks.
- Initially a transaction is in growing phase and acquires lock as needed and in between can perform operation reach to lock point and once a transaction releases a lock, it can issue no more lock requests i.e. it enters the shrinking phase.

.

T1	T2
LOCK-X(A)	
READ(A)	
WRITE(A)	
	LOCK-S(B)
	READ(B)
LOCK-X(B)	
READ(B)	
WRITE(B)	
	LOCK-S(A)
	READ(A)
	UNLOCK(B)
UNLOCK(A)	
UNLOCK(B)	
	UNLOCK(A)

Properties

2PL ensures conflict serializability, and the ordering of transaction over lock points is itself a serializability order of a schedule in 2PL.

If a schedule is allowed in 2PL protocol then definitely it is always conflict serializable. But it is not necessary that if a schedule is conflict serializable then it will be generated by 2pl. Equivalent serial schedule is based on the order of lock points.

View serializability is also guaranteed.

Does not ensure freedom from deadlock

May cause non-recoverability.

Cascading rollback may occur.

S2

T1	T2
LOCK-X(A)	
READ(A)	
WRITE(A)	
UNLOCK(A)	
	LOCK-S(A)
	READ(A)
	Commit
Commit	

Q Consider the following two statements about database transaction schedules:
I. Strict two-phase locking protocol generates conflict serializable schedules that are also recoverable.
II. Timestamp-ordering concurrency control protocol with Thomas Write Rule can generate view serializable schedules that are not conflict serializable.
Which of the above statements is/are TRUE? **(GATE-2019) (1 Marks)**

(a) Both I and II
(b) Neither I nor II
(c) II only
(d) I only

Ans: a

Q Which of the following concurrency control protocols ensure both conflict serializability and freedom from deadlock? **(GATE-2010) (2 Marks)**

I. 2-phase locking
II. Time-stamp ordering

(A) I only **(B)** II only **(C)** Both I and II **(D)** Neither I nor II

Answer: (B)

Q Which of the following concurrency protocol ensures both conflict serializability and freedom from deadlock? **(NET-JUNE-2015)**

(a) 2 - phase Locking
(b) Time stamp - ordering
(1) Both (a) and (b)
(2) (a) only
(3) (b) only
(4) Neither (a) nor (b)

Ans. 3

Q Which of the following statements is wrong? **(NET-JUNE-2007)**

(A) 2-phase Locking Protocols suffer from deadlocks
(B) Time-Stamp Protocols suffer from more aborts
(C) Time-Stamp Protocols suffer from cascading roll back where as 2-Phase locking Protocol do not
(D) None of these

Q Consider the following two-phase locking protocol. Suppose a transaction T accesses (for read or write operations), a certain set of objects$\{O_1,...,O_k\}$. This is done in the following manner:

Step1. T acquires exclusive locks to $O_1,...,O_k$ in increasing order of their addresses.
Step2. The required operations are performed.
Step3. All locks are released.

This protocol will (GATE- 2016) (1 Marks)

(a) guarantee serializability and deadlock-freedom
(b) guarantee neither serializability nor deadlock-freedom
(c) guarantee serializability but not deadlock-freedom
(d) guarantee deadlock-freedom but not serializability

Ans: a

Q Two phase protocol in a database management system is: **(NET-DEC-2006)**

(A) a concurrency mechanism that is not deadlock free
(B) a recovery protocol used for restoring a database after a crash
(C) Any update to the system log done in 2-phases
(D) not effective in Database

Ans: a

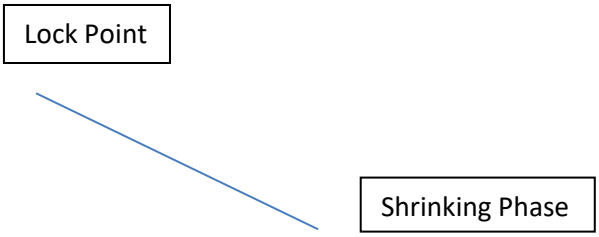
Q Which of the following is correct? **(NET-DEC-2014)**
I. Two phase locking is an optimistic protocol.
II. Two phase locking is pessimistic protocol
III. Time stamping is an optimistic protocol.
IV. Time stamping is pessimistic protocol.
(A) I and III **(B)** II and IV **(C)** I and IV **(D)** II and III
Ans: b

Q Which of the following concurrency protocol ensures both conflict serializability and freedom from deadlock: **(NET-JUNE-2014)**
I. 2-phase locking **II.** Time phase ordering
(A) Both I & II **(B)** II only **(C)** I only **(D)** Neither I nor II
Ans: b

Q Which of the following time stamp ordering protocol(s) allow(s) the following schedules?
T: $W_1(A)$; $W_2(A)$; $W_3(A)$; $R_2(A)$; $R_4(4)$;
Time stamps: $T_1 : 5$, $T_2 : 10$, $T_3 : 15$; $T_4 : 20$
a) Thomas write rule b) Basic time stamp

Variants of Two- Phase locking method

- Different variants of 2pl are used where we try ensure the properties like deadlock, recoverable, cascade less.
- Conservative 2pl
- The idea is there is no growing phase transaction start directly from lock point, i.e. transaction must first acquire all the required locks then only it can start execution. If all the locks are not available then transaction must release the acquired locks and must wait.
    - Shrinking phase will work as usual, and transaction can unlock any data item anytime.
    - we must have a knowledge in future to understand what is data required so that we can use it



- Properties
- Conflict serializable, view serializable, Independence from deadlock
    - Still have possibility of irrecoverable schedule and cascading rollbacks.

Q In conservative two-phase locking protocol, a transaction
a) Should release exclusive locks only after the commit operation
b) Should release all the locks only at beginning of the transaction
c) should acquire all the locks at beginning of the transaction
d) Should acquire all the exclusive locks at beginning transaction

RIGOROUS 2PL

- requires that all locks be held until the transaction commits.
- This protocol requires that locking be two phase and also all the locks taken be held by transaction until that transaction commit.
- Hence there is no shrinking phase in the system.

E.g.

T1	T2
$X(A)$	
$R(A)$	
$W(A)$	
$X(B)$	
$R(B)$	
$W(B)$	
Commit	
	$X(A)$
	$R(A)$
	$W(A)$
	$X(B)$
	$R(B)$
	$W(B)$
	Commit

- Properties
- Conflict serializable, view serializable, recoverable and cascade less
  - Still have possibility of deadlocks.

Q In a Rigorous 2 phase protocol

a) All shared locks held by the transaction are related after the transaction is committed

b) All exclusive locks held by the transaction are released after the transaction is committed

c) All locks held by the transaction are released after the transaction is committed

d) All locks held by the transaction are released before the transaction is committed

STRICT 2PL

- that all exclusive-mode locks taken by a transaction be held until that transaction commits. This requirement ensures that any data written by an uncommitted transaction are locked in exclusive mode until the transaction commits, preventing any other transaction from reading the data.
- This protocol requires that locking be two phase and also that exclusive –mode locks taken by transaction be held until that transaction commits.
- So it is simplified form of rigorous 2pl

- ***E.g.***

Locking (Strict 2PL)

T1	T2
$S(A)$	
$R(A)$	
	$S(A)$
	$R(A)$
	$X(B)$
	$R(B)$
	$W(B)$
	Commit
$X(C)$	
$R(C)$	
$W(C)$	
Commit	

Schedule Following Strict 2PL with Interleaved Actions

- It ensures serializability (Equivalent serial schedule order based on the order of lock points.
- Ensures strict recoverability.
- Deadlock still possible in strict 2PL.
- In general, deadlocks are a necessary evil associated with locking, if we want to avoid inconsistent states. Deadlocks are definitely preferable to inconsistent states, since they can be handled by rolling back transactions, whereas inconsistent states may lead to real-world problems that cannot be handled by the database system.
- Strict two-phase locking and rigorous two-phase locking (with lock conversions) are used extensively in commercial database systems.

Q Which of the following is a false statement?
a) A schedule which is allowed under basic 2PL is always under strict 2PL
b) A schedule which is allowed under strict 2PL is always allowed under basic 2PL
c) A schedule which is allowed under basic time stamp protocol is always allowed under Thomas write rule
d) None of these

Q Which of the following statement is/are correct

a) Every conflict serializable schedule allowed under 2PL protocol is allowed by basic time stamping protocol.

b) Every schedule allowed under basic time stamping protocol is allowed by Thomas-write rule

c) Every schedule allowed under Thomas-write rule is allowed by basic time stamping protocol

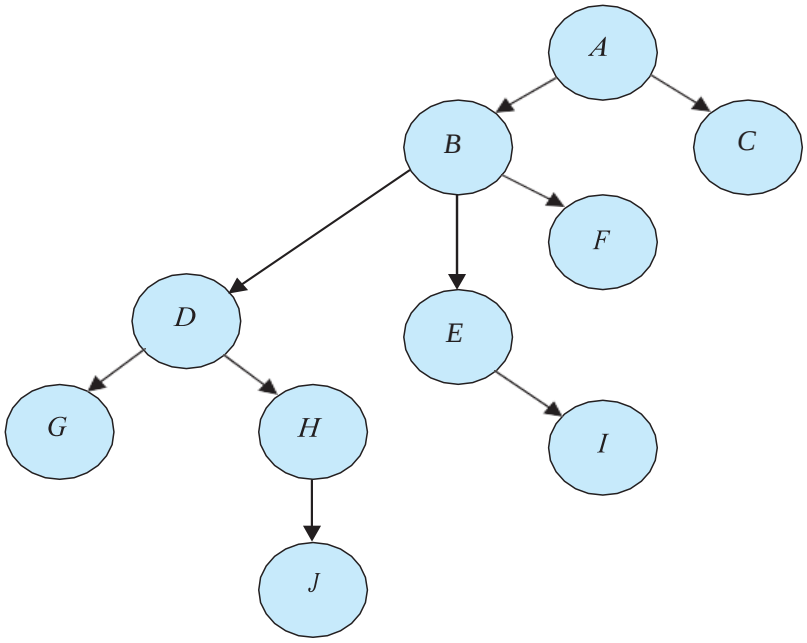
d) none

Graph based protocol

- if we wish to develop protocols that are not two phase, we need additional information on how each transaction will access the database.
- There are various models that can give us the additional information, each differing in the amount of information provided.
- The simplest model requires that we have prior knowledge about the order in which the database items will be accessed.
- Given such information, it is possible to construct locking protocols that are not two phases, but that, nevertheless, ensure conflict serializability.
- To acquire such prior knowledge, we impose a partial ordering → on the set **D** = $\{d_1, d_2, \ldots, d_h\}$ of all data items. If $d_i \rightarrow d_j$, then any transaction accessing both d_i and d_j must access d_i before accessing d_j.
- This partial ordering may be the result of either the logical or the physical organization of the data, or it may be imposed solely for the purpose of concurrency control.
- The partial ordering implies that the set **D** may now be viewed as a directed acyclic graph, called a **database graph**.
- Here for the sake of simplicity, we will follow two restriction
    - Will study graphs that are rooted trees.
    - Will restrict to employ only *exclusive* locks.

Tree Protocol

- In the tree protocol, the only lock instruction allowed is lock-X. Each transaction Ti can lock a data item at most once, and must observe the following rules:
- The first lock by Ti may be on any data item.
- Subsequently, a data item Q can be locked by Ti only if the parent of Q is currently locked by Ti.
- Data items may be unlocked at any time.



- A data item that has been locked and unlocked by Ti cannot subsequently be relocked by Ti.

T_1: lock-X(*B*); lock-X(*E*); lock-X(*D*); unlock(*B*); unlock(*E*); lock-X(*G*); unlock(*D*); unlock(*G*).
T_2: lock-X(*D*); lock-X(*H*); unlock(*D*); unlock(*H*).
T_3: lock-X(*B*); lock-X(*E*); unlock(*E*); unlock(*B*).
T_4: lock-X(*D*); lock-X(*H*); unlock(*D*); unlock(*H*).

Properties

- All schedules that are legal under the tree protocol are conflict serializable.
- tree protocol ensures freedom from deadlock.
- tree protocol does not ensure recoverability and cascadelessness.
- The tree-locking protocol has another advantage over the two-phase locking protocol in that unlocking may occur earlier. Earlier unlocking may lead to shorter waiting times, and to an increase in concurrency.
- A transaction may have to lock data items that it does not access. This additional locking results in increased locking overhead, the possibility of additional waiting time, and a potential decrease in concurrency.
- Without prior knowledge of what data items will need to be locked, transactions will have to lock the root of the tree, and that can reduce concurrency greatly.
- there may be conflict-serializable schedules that cannot be obtained through the tree protocol. Indeed, there are schedules possible under the two-phase locking protocol that are not possible under the tree protocol, and vice versa.
- To ensure recoverability and cascadelessness, the protocol can be modified to not permit release of exclusive locks until the end of the transaction. Holding exclusive locks until the end of the transaction reduces concurrency.

Deadlock Handling

- There are two principal methods for dealing with the deadlock problem.
- We can use a **deadlock prevention** protocol to ensure that the system will *never* enter a deadlock state, Prevention is commonly used if the probability that the system would enter a deadlock state is relatively high
- Alternatively, we can allow the system to enter a deadlock state, and then try to recover by using a **deadlock detection** and **deadlock recovery** scheme.
- both methods may result in transaction rollback more efficient. Note that a detection and recovery scheme require overhead that includes not only the run-time cost of maintaining the necessary information and of executing the detection algorithm, but also the potential losses inherent in recovery from a deadlock.

Deadlock Prevention

- One approach ensures that no hold & waits can occur it is the simplest scheme requires that each transaction locks all its data items before it begins execution, either all are locked in one step or none are locked e.g. conservative 2PL.
- Other approach ensures that no cyclic waits can occur by ordering the requests for locks, i.e. is to impose an ordering of all data items, and to require that a transaction lock data items only in a sequence consistent with the ordering e.g. tree protocol.
    - it is often hard to predict, before the transaction begins, what data items need to be locked;
    - data-item utilization may be very low, since many of the data items may be locked but unused for a long time.
- The other approach is closer to deadlock recovery, and performs transaction rollback instead of waiting for a lock, whenever the wait could potentially result in a deadlock i.e. to use preemption and transaction rollbacks.

- So when a transaction T_j requests a lock that transaction T_i holds, the lock granted to T_i may be **preempted** by rolling back of T_i, and granting of the lock to T_j. To control the preemption, we assign a unique timestamp, to each transaction when it begins. The system uses these timestamps only to decide whether a transaction should wait or roll back. Locking is still used for concurrency control. If a transaction is rolled back, it retains its *old* timestamp when restarted. Two different deadlock-prevention schemes using timestamps have been proposed:
    - The **wait–die** scheme is a non-preemptive technique. 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 (that is, T_i is older than T_j). Otherwise, T_i is rolled back (dies).
    - The **wound–wait** scheme is a preemptive technique. It is a counterpart to the wait–die scheme. 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 larger than that of T_j (that is, T_i is younger than T_j). Otherwise, T_j is rolled back (T_j is *wounded* by T_i).
- The major problem with both of these schemes is that unnecessary rollbacks may occur. Another simple approach to deadlock prevention is based on **lock timeouts**. In this approach, a transaction that has requested a lock waits for at most a specified amount of time. If the lock has not been granted within that time, the transaction is said to time out, and it rolls itself back and restarts. If there was in fact a deadlock, one or more transactions involved in the deadlock will time out and roll back, allowing the others to proceed. This scheme falls somewhere between deadlock prevention, where a deadlock will never occur, and deadlock detection and recovery.
- The timeout scheme is particularly easy to implement, and works well if transactions are short and if long waits are likely to be due to deadlocks. However, in general it is hard to decide how long a transaction must wait before timing out. Too long a wait results in unnecessary delays once a deadlock has occurred. Too short a wait results in transaction rollback even when there is no deadlock, leading to wasted resources. Starvation is also

a possibility with this scheme. Hence, the timeout-based scheme has limited applicability.

Deadlock Detection and Recovery

- If a system does not employ some protocol that ensures deadlock freedom, then a detection and recovery scheme must be used. An algorithm that examines the state of the system is invoked periodically to determine whether a deadlock has occurred. If one has, then the system must attempt to recover from the deadlock. To do so, the system must:
- Maintain information about the current allocation of data items to transaction, as well as any outstanding data item requests.
- Provide an algorithm that uses this information to determine whether the system has entered a deadlock state.
- Recover from the deadlock when the detection algorithm determines that a deadlock exists.

- Deadlock Detection - Deadlocks can be described precisely in terms of a directed graph called a **wait- for graph**. This graph consists of a pair $G = (V, E)$, where V is a set of vertices and E is a set of edges. The set of vertices consists of all the transactions in the system. Each element in the set E of edges 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 transaction T_i to T_j, implying that transaction T_i is waiting for transaction T_j to release a data item that it needs.
- A deadlock exists in the system if and only if the wait-for graph contains a cycle. Each transaction involved in the cycle is said to be deadlocked. To detect deadlocks, the system needs to maintain the wait-for graph, and periodically to invoke an algorithm that searches for a cycle in the graph.
- When should we invoke the detection algorithm? The answer depends on two factors:
- How often does a deadlock occur? - If deadlocks occur frequently, then the detection algorithm should be invoked more frequently. Data items allocated to deadlocked transactions will be unavailable to other transactions until the deadlock can be broken.
- How many transactions will be affected by the deadlock? - In addition, the number of cycles in the graph may also grow. In the worst case, we would invoke the detection algorithm every time a request for allocation could not be granted immediately.

Recovery from Deadlock

- When a detection algorithm determines that a deadlock exists, the system must **recover** from the deadlock. The most common solution is to roll back one or more transactions to break the deadlock. Three actions need to be taken:
- **Selection of a victim**. Given a set of deadlocked transactions, we must determine which transaction (or transactions) to roll back to break the deadlock. We should roll back those transactions that will incur the minimum cost. Unfortunately, the term *minimum cost* is not a precise one. Many factors may determine the cost of a rollback, including:
- How long the transaction has computed, and how much longer the transaction will compute before it completes its designated task.
- How many data items the transaction has used.
- How many more data items the transaction needs for it to complete.
- How many transactions will be involved in the rollback.
- **Rollback**. Once we have decided that a particular transaction must be rolled back, we must determine how far this transaction should be rolled back.
- The simplest solution is a **total rollback**: Abort the transaction and then restart it.
- However, it is more effective to roll back the transaction only as far as necessary to break the deadlock. Such **partial rollback** requires the system to maintain additional information about the state of all the running transactions. Specifically, the sequence of lock requests/grants and updates performed by the transaction needs to be recorded. The deadlock detection mechanism should decide which locks the selected transaction needs to release in order to break the deadlock. The selected transaction must be rolled back to the point where it obtained the first of these locks, undoing all actions it took after that point. The recovery mechanism must be capable of performing such partial rollbacks. Furthermore, the transactions must be capable of resuming execution after a partial rollback.
- **Starvation**. In a system where the selection of victims is based primarily on cost factors, it may happen that the same transaction is always picked as a victim. As a result, this transaction never completes its designated task, thus there is **starvation**. We must ensure that a transaction can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

Multiple Granularity

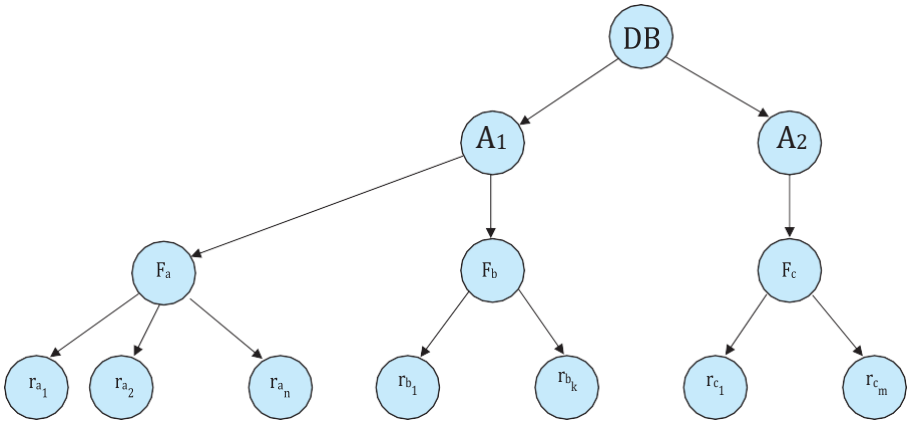
In the concurrency-control schemes described thus far, we have used each individual data item as the unit on which synchronization is performed.

There are circumstances, however, where it would be advantageous to group several data items, and to treat them as one individual synchronization unit. For example, if a transaction T_i needs to access the entire database, and a locking protocol is used, then T_i must lock each item in the database. Clearly, executing these locks is time-consuming. It would be better if T_i could issue a *single* lock request to lock the entire database. On the other hand, if transaction T_j needs to access only a few data items, it should not be required to lock the entire database, since otherwise concurrency is lost.

What is needed is a mechanism to allow the system to define multiple levels of **granularity**. This is done by allowing data items to be of various sizes and defining a hierarchy of data granularities, where the small granularities are nested within larger ones. Such a hierarchy can be represented graphically as a tree. A non-leaf node of the multiple-granularity tree represents the data associated with its descendants. In the tree protocol, each node is an independent data item.

As an illustration, consider the tree of Figure, which consists of four levels of nodes. The highest level represents the entire database. Below it are nodes of type *area*; the database consists of exactly these areas. Each area in turn has nodes of type *file* as its children. Each area contains exactly those files that are its child nodes. No file is in more than one area. Finally, each file has nodes of type *record*. As before, the file consists of exactly those records that are its child nodes, and no record can be present in more than one file.

Each node in the tree can be locked individually. As we did in the two-phase locking protocol, we shall use **shared** and **exclusive** lock modes. When a transaction locks a node, in either shared or exclusive mode, the transaction also has implicitly locked all the descendants of that node in the same lock mode. For example, if transaction T_i gets an **explicit lock** on file F_c of Figure, in exclusive mode, then it has an **implicit lock** in exclusive mode on all the records belonging to that file. It does not need to lock the individual records of F_c explicitly. explicitly.



Suppose now that transaction T_k wishes to lock the entire database. To do so, it simply must lock the root of the hierarchy. Note, however, that T_k should not succeed in locking the root node, since T_i is currently holding a lock on part of the tree (specifically, on file F_b). But how does the system determine if the root node can be locked? One possibility is for it to search the entire tree. This solution, however, defeats the whole purpose of the multiple-granularity locking scheme. A more efficient way to gain this knowledge is to introduce a new class of lock modes, called **intention lock modes**. If a node is locked in an intention mode, explicit locking is done at a lower level of the tree (that is, at a finer granularity). Intention locks are put on all the ancestors of a node before that node is locked explicitly. Thus, a transaction does not need to search the entire tree to determine whether it can lock a node successfully. A transaction wishing to lock a node—say, Q—must traverse a path in the tree from the root to Q. While traversing the tree, the transaction locks the various nodes in an intention mode.

There is an intention mode associated with shared mode, and there is one with exclusive mode. If a node is locked in **intention-shared (IS) mode**, explicit locking is being done at a lower level of the tree, but with only shared-mode locks. Similarly, if a node is locked in **intention-exclusive (IX) mode**, then explicit locking is being done at a lower level, with exclusive-mode or shared-mode locks. Finally, if a node is locked in **shared and intention-exclusive (SIX) mode**, the subtree rooted by that node is locked explicitly in shared mode, and that explicit locking is being done at a lower level with exclusive-mode locks.

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

The **multiple-granularity locking protocol** uses these lock modes to ensure serializability. It requires that a transaction T_i that attempts to lock a node Q must follow these rules:

1. Transaction T_i must observe the lock-compatibility function of Figure 15.16.
2. Transaction T_i must lock the root of the tree first, and can lock it in any mode.
3. Transaction T_i can lock a node Q in S or IS mode only if T_i currently has the parent of Q locked in either IX or IS mode.
4. Transaction T_i can lock a node Q in X, SIX, or IX mode only if T_i currently has the parent of Q locked in either IX or SIX mode.

5. Transaction T_i can lock a node only if T_i has not previously unlocked any node (that is, T_i is two phase).
6. Transaction T_i can unlock a node Q only if T_i currently has none of the children of Q locked.

Observe that the multiple-granularity protocol requires that locks be acquired in *top-down* (root-to-leaf) order, whereas locks must be released in *bottom-up* (leaf- to-root) order. This protocol enhances concurrency and reduces lock overhead. It is particularly useful in applications that include a mix of:

- Short transactions that access only a few data items.
- Long transactions that produce reports from an entire file or set of files.

Validation-Based Protocols

In cases where a majority of transactions are read-only transactions, the rate of conflicts among transactions may be low. Thus, many of these transactions, if executed without the supervision of a concurrency-control scheme, would nevertheless leave the system in a consistent state.

A concurrency-control scheme imposes overhead of code execution and possible delay of transactions. It may be better to use an alternative scheme that imposes less overhead.

The **validation protocol** requires that each transaction T_i executes in two or three different phases in its lifetime, depending on whether it is a read-only or an update transaction. The phases are, in order:

Read phase. During this phase, the system executes transaction T_i. It reads the values of the various data items and stores them in variables local to Ti. It performs all write operations on temporary local variables, without updates of the actual database.

Validation phase. The validation test (described below) is applied to transaction T_i. This determines whether T_i is allowed to proceed to the write phase without causing a violation of serializability. If a transaction fails the validation test, the system aborts the transaction.

Write phase. If the validation test succeeds for transaction T_i, the temporary local variables that hold the results of any write operations performed by T_i are copied to the database.

To perform the validation test, we need to know when the various phases of transactions took place. We shall, therefore, associate three different timestamps with each transaction T_i:

1. **Start**(T_i), the time when T_i started its execution.
2. **Validation**(T_i), the time when T_i finished its read phase and started its validation phase.
3. **Finish**(T_i), the time when T_i finished its write phase.

We determine the serializability order by the timestamp-ordering technique, using the value of the timestamp Validation(T_i). Thus, the value TS(T_i) = Validation(T_i) and, if TS(T_j) < TS(T_k), then any produced schedule must be equivalent to a serial schedule in which transaction T_j appears before transaction T_k. The reason we have chosen Validation(T_i), rather than Start(T_i), as the timestamp of transaction T_i is that we can expect faster response time provided that conflict rates among transactions are indeed low.

The **validation test** for transaction T_i requires that, for all transactions T_k with TS(T_k) < TS(T_i), one of the following two conditions must hold:

1. Finish(T_k) < Start(T_i). Since T_k completes its execution before T_i started, the

serializability order is indeed maintained.

2. The set of data items written by T_k does not intersect with the set of data items read by T_i, and T_k completes its write phase before T_i starts its validation phase (Start(T_i) < Finish(T_k) < Validation(T_i)). This condition ensures that the writes of T_k and T_i do not overlap. Since the writes of T_k do not affect the read of T_i, and since T_i cannot affect the read of T_k, the serializability order is indeed maintained.

The validation scheme automatically guards against cascading rollbacks, since the actual writes take place only after the transaction issuing the write has committed. However, there is a possibility of starvation of long transactions, due to a sequence of conflicting short transactions that cause repeated restarts of the long transaction. To avoid starvation, conflicting transactions must be temporarily blocked, to enable the long transaction to finish.

This validation scheme is called the **optimistic concurrency-control** scheme since transactions execute optimistically, assuming they will be able to finish execution and validate at the end. In contrast, locking and timestamp ordering are pessimistic in that they force a wait or a rollback whenever a conflict is detected, even though there is a chance that the schedule may be conflict serializable.

When several transactions execute concurrently in the database, the consistency of data may no longer be preserved. It is necessary for the system to control the interaction among the concurrent transactions, and this control is achieved through one of a variety of mechanisms called *concurrency-control* schemes.

To ensure serializability, we can use various concurrency-control schemes. All these schemes either delay an operation or abort the transaction that issued the operation. The most common ones are locking protocols, timestamp- ordering schemes, validation techniques, and multiversion schemes.

- A locking protocol is a set of rules that state when a transaction may lock and unlock each of the data items in the database.
- The two-phase locking protocol allows a transaction to lock a new data item only if that transaction has not yet unlocked any data item. The protocol ensures serializability, but not deadlock freedom. In the absence of information concerning the manner in which data items are accessed, the two-phase locking protocol is both necessary and sufficient for ensuring serializability.
- The strict two-phase locking protocol permits release of exclusive locks only at the end of transaction, in order to ensure recoverability and cascadelessness

of the resulting schedules. The rigorous two-phase locking protocol releases all locks only at the end of the transaction.

- Graph-based locking protocols impose restrictions on the order in which items are accessed, and can thereby ensure serializability without requiring the use of two-phase locking, and can additionally ensure deadlock freedom.
- Various locking protocols do not guard against deadlocks. One way to pre- vent deadlock is to use an ordering of data items, and to request locks in a sequence consistent with the ordering.
- Another way to prevent deadlock is to use preemption and transaction roll- backs. To control the preemption, we assign a unique timestamp to each transaction. The system uses these timestamps to decide whether a transac- tion should wait or roll back. If a transaction is rolled back, it retains its old timestamp when restarted. The wound–wait scheme is a preemptive scheme.
- If deadlocks are not prevented, the system must deal with them by using a deadlock detection and recovery scheme. To do so, the system constructs a wait-for graph. A system is in a deadlock state if and only if the wait-for graph contains a cycle. When the deadlock detection algorithm determines that a deadlock exists, the system must recover from the deadlock. It does so by rolling back one or more transactions to break the deadlock.
- There are circumstances where it would be advantageous to group several data items, and to treat them as one aggregate data item for purposes of working, resulting in multiple levels of granularity. We allow data items of various sizes, and define a hierarchy of data items, where the small items are nested within larger ones. Such a hierarchy can be represented graphically as a tree. Locks are acquired in root-to-leaf order; they are released in leaf-to-root order. The protocol ensures

serializability, but not freedom from deadlock.

- A timestamp-ordering scheme ensures serializability by selecting an ordering in advance between every pair of transactions. A unique fixed timestamp is associated with each transaction in the system. The timestamps of the transactions determine the serializability order. Thus, if the timestamp of transaction T_i is smaller than the timestamp of transaction T_j, then the scheme ensures that the produced schedule is equivalent to a serial schedule in which transaction T_i appears before transaction T_j. It does so by rolling back a transaction whenever such an order is violated.
- A validation scheme is an appropriate concurrency-control method in cases where a majority of transactions are read-only transactions, and thus the rate of conflicts among these transactions is low. A unique fixed timestamp is associated with each transaction in the system. The serializability order is determined by the timestamp of the transaction. A transaction in this scheme is never delayed. It must, however, pass a validation test to complete. If it does not pass the validation test, the system rolls it back to its initial state.
- A multiversion concurrency-control scheme is based on the creation of a new version of a data item for each transaction that writes that item. When a read operation is issued, the system selects one of the versions to be read. The concurrency-control scheme ensures that the version to be read is selected in a manner that ensures serializability, by using timestamps. A read operation always succeeds.

  In multiversion timestamp ordering, a write operation may result in the rollback of the transaction.
  In multiversion two-phase locking, write operations may result in a lock wait or, possibly, in deadlock.

- Snapshot isolation is a multiversion concurrency-control protocol based on validation, which, unlike multiversion two-phase locking, does not require transactions to be declared as read-only or update. Snapshot isolation does not guarantee serializability, but is nevertheless supported by many database systems.
- A **delete** operation may be performed only if the transaction deleting the tuple has an exclusive lock on the tuple to be deleted. A transaction that inserts a new tuple into the database is given an exclusive lock on the tuple.
- Insertions can lead to the phantom phenomenon, in which an insertion log- ically conflicts with a query even though the two transactions may access no tuple in common. Such conflict cannot be detected if locking is done only on tuples accessed by the transactions. Locking is required on the data used to find the tuples in the relation. The index-locking technique solves this problem by requiring locks on certain index nodes. These locks ensure that all conflicting transactions conflict on a real data item, rather than on a phantom.
- Weak levels of consistency are used in some applications where consistency of query results is not critical, and using serializability would result in queries adversely affecting transaction processing. Degree-two consistency is one such weaker level of

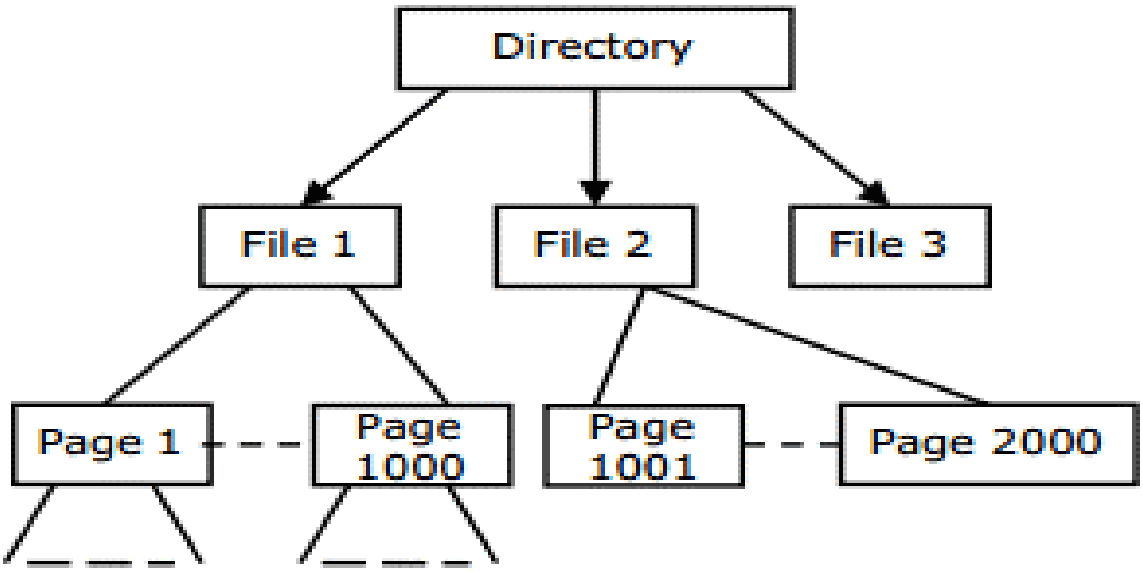
consistency; cursor stability is a special case of degree- two consistency, and is widely used.

- Concurrency control is a challenging task for transactions that span user interactions. Applications often implement a scheme based on validation of writes using version numbers stored in tuples; this scheme provides a weak level of serializability, and can be implemented at the application level without modifications to the database.
- Special concurrency-control techniques can be developed for special data structures. Often, special techniques are applied in B^+-trees to allow greater concurrency. These techniques allow non-serializable access to the B^+-tree, but they ensure that the B^+-tree structure is correct, and ensure that accesses to the database itself are serializable.

Q If the interference among the transactions is less, which of the following concurrency control techniques have less overloaded in the execution?

a) Locking techniques
b) Time stamping techniques
c) Validation technique
d) None

Q Consider a directory having 3 files, each file has 1000 pages and each page has 100 records



In multiple granularity locking protocol, if a transaction reads records from page 200 to page 700. What is the sequence of locks acquired?

a) IX on directory, S on file 1
b) IS on directory, S on file 1
c) IS on directory, X on file 1
d) IS on directory, S on file 2

Q _____ rules used to limit the volume of log information that has to be handled and

processed in the event of system failure involving the loss of volatile information. **(NET-DEC-2014)**

(A) Write-ahead log
(B) Check-pointing
(C) Log buffer
(D) Thomas

Ans: b

Q Immediate updates as a recovery protocol is preferable, when: **(NET-JUNE-2006)**

(A) Database reads more than writes
(B) Writes are more than reads
(C) It does not matter as it is good in both the situations
(D) There are only writes

Ans: b

Q In DBMS, deferred update means: **(NET-DEC-2006)**

(A) All the updates are done first but the entries are made in the log file later
(B) All the log files entries are made first but the actual updates are done later
(C) Every update is done first followed by a writing on the log file
(D) Changes in the views are deferred till a query asks for a view

Ans: b

Q Which of the following statement is true? **(NET-DEC-2009)**

I. 2-phase locking protocol suffer from dead lock.
II. Time stamp protocol suffer from more aborts.
III. A block hole in a DFD is a data store with only inbound flows.
IV. Multivalued dependency among attribute is checked at 3 NF level.
V. An entity-relationship diagram is a tool to represent event model.

(A) I, II, II **(B)** II, III, IV **(C)** III, IV, V **(D)** II, IV, V

Ans: a

Q Which of the following is an optimistic concurrency control method? **(NET-DEC-2010)**

(A) Validation based
(B) Time stamp ordering
(C) Lock-based
(D) None of these

Ans: a

Q The basic variants of time-stamp based method of concurrency control are **(NET-JUNE-2011)**

(A) Total time stamp-ordering
(B) Partial time stamp ordering
(C) Multiversion Time stamp ordering
(D) All of the above

Ans: d

Q Which of the following is the recovery management technique in DDBMS? **(NET-JUNE-2011)**

(A) 2PC (Two Phase Commit)
(B) Backup
(C) Immediate update
(D) All of the above

Ans: d

Q Match the following: **(NET-JUNE-2014)**

List – I	List – II
a. Timeout ordering protocol	i. Wait for graph
b. Deadlock prevention	ii. Roll back
c. Deadlock Detection	iii. Wait-die scheme
d. Deadlock recovery	iv. Thomas Write rule

Codes:

	a	**b**	**c**	**d**
(a)	iv	iii	i	ii
(b)	iii	ii	iv	i
(c)	ii	i	iv	iii
(d)	iii	i	iv	iii

Ans: a