

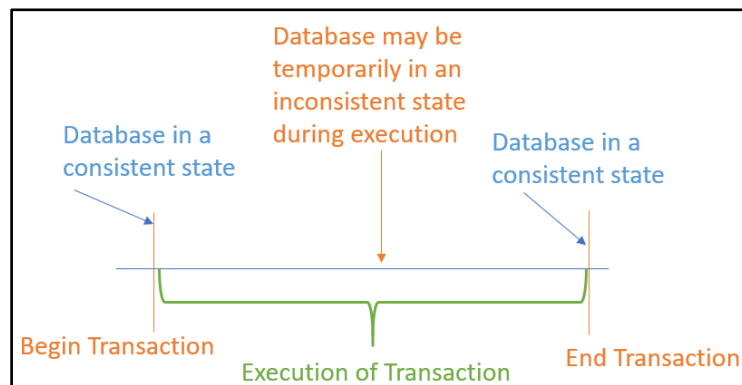
6. Transaction & Recovery Management

Objectives:

1. Transaction Concepts
2. ACID Properties
3. Problems during Concurrent Transactions
4. Serializability
5. Types of Schedules
6. States of Transaction
7. Lock Management
8. Locking Protocols (2PL, Multiple Granularity, 2PC, Timestamp ordering)
9. Deadlock Handling
10. Log-based Recovery

Transaction Concepts

Transaction: Transaction is a set of DML operations that performs a Business (/single logical) unit of work.



The transaction consists of all operations executed between the **begin transaction** and **end transaction**. We require that the database system maintain the following properties (**ACID** properties) of the transactions:

1. Atomicity
2. Consistency
3. Isolation
4. Durability

Transaction (Txn) begins with the execution of **first DML operation** and ends with **COMMIT** or **ROLLBACK** or **DDL** or **TCL**.

Types of Transaction:

1. READ only
2. READ WRITE (Default option)

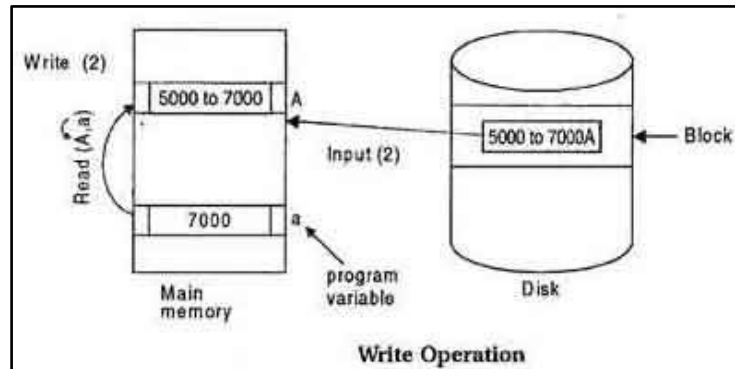
Transactions access data using two operations:

1. READ(X):

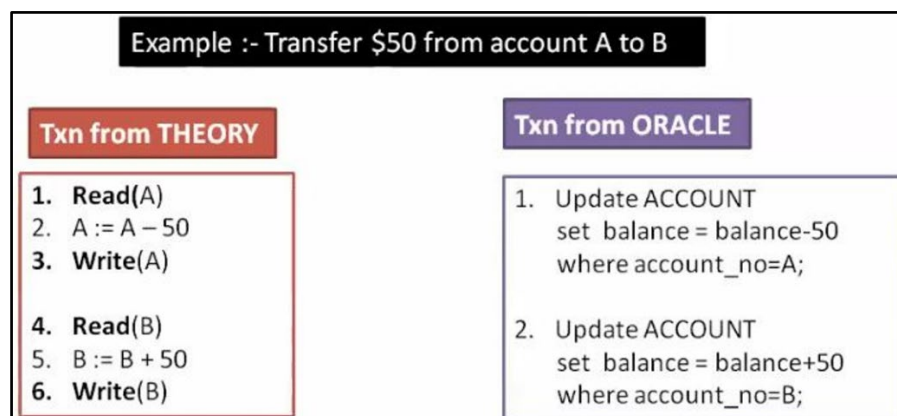
- Find the address of disk block containing item X.
- Copy that disk block into a buffer within Main memory.
- Copy item X into a program variable (X) from buffer.

2. WRITE(X):

- Copy item X from program variable to its location in buffer.
- Stores the updated block from buffer to disk.



Example of Transaction:



ACID Properties

Every transaction must ensure that the following properties will be satisfied:

Atomicity:

Suppose that, just before the execution of transaction T_i , the values of accounts A and B are \$1000 and \$2000, respectively. Now suppose that, during the execution of transaction T_i , a failure occurs that prevents T_i from completing its execution successfully. Further, suppose that the failure happened after the write(A) operation but before the write(B) operation. In this case, the values of accounts A and B reflected in the database are \$950 and \$2000. The system destroyed \$50 as a result of this failure. In particular, we note that the sum $A + B$ is no longer preserved.

Thus, because of the failure, the state of the system no longer reflects a real state of the world that the database is supposed to capture. We term such a state an **inconsistent state**. We must ensure that such inconsistencies are not visible in a database system. Note, however, that the system must at some point be in an inconsistent state. Even if transaction T_i is executed to completion, there exists a point at which the value of account A is \$950 and the value of account B is \$2000, which is clearly an inconsistent state. This state, however, is eventually replaced

by the consistent state where the value of account A is \$950, and the value of account B is \$2050. Thus, if the transaction never started or was guaranteed to complete, such an inconsistent state would not be visible except during the execution of the transaction. That is the reason for the atomicity requirement: If the atomicity property is present, all actions of the transaction are reflected in the database, or none are.

The basic idea behind ensuring atomicity is this: The database system keeps track (on disk) of the old values of any data on which a transaction performs a write. This information is written to a file called the *log*. If the transaction does not complete its execution, the database system restores the old values from the log to make it appear as though the transaction never executed.

Consistency:

The consistency requirement here is that the sum of A and B be unchanged by the execution of the transaction and the database must be in consistent state before and after execution of transaction (It may be temporarily inconsistent during the transaction). For example, if a transaction transfers 50\$ from account A to B , where A has balance 100\$ and B has 500\$, so before transaction sum is 600\$, therefore after transaction sum must be 600\$ (i.e., balance in A 50\$ and in B 550\$). Without the consistency requirement, money could be created or destroyed by the transaction! It can be verified easily that, if the database is consistent before an execution of the transaction, the database remains consistent after the execution of the transaction.

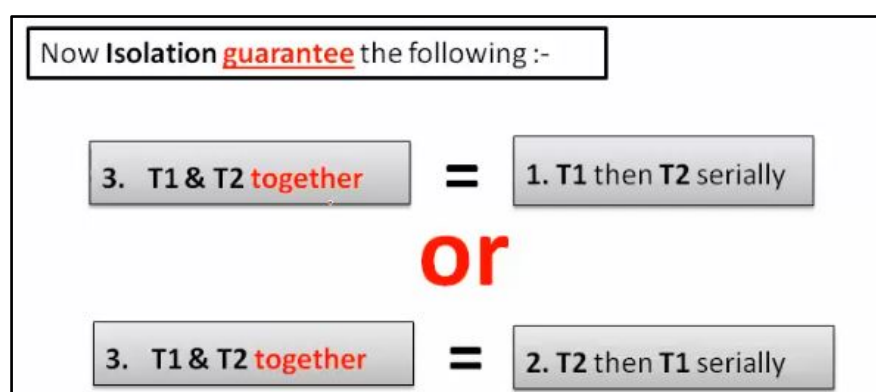
Ensuring consistency for an individual transaction is the responsibility of the application programmer who codes the transaction. This task may be facilitated by automatic testing of integrity constraints.

Isolation:

When transaction occurs concurrently, the net effect must be identical to one of the possible serial execution. Even if the consistency and atomicity properties are ensured for each transaction, if several transactions are executed concurrently, their operations may interleave in some undesirable way, resulting in an inconsistent state.

For example, as we saw earlier, the database 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 point 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 database may be left in an inconsistent state even after both transactions have completed.

A way to avoid the problem of concurrently executing transactions is to execute transactions serially - that is, one after the other. However, concurrent execution of transactions provides significant performance benefits.



Durability:

Once the execution of the transaction completes successfully, and the user who initiated the transaction has been notified that the transfer of funds has taken place, it must be the case that no system failure can result in a loss of data corresponding to this transfer of funds. The durability property guarantees that, once a transaction completes successfully, all the updates that it carried out on the database persist, even if there is a system failure or disk failure or hardware failure after the transaction completes execution.


We can guarantee durability by ensuring that either:

1. The updates carried out by the transaction have been written to disk before the transaction completes.
2. Information about the updates carried out by the transaction and written to disk is sufficient to enable the database to reconstruct the updates when the database system is restarted after the failure.

Problems during Concurrent Transactions

Transaction-processing systems usually allow multiple transactions to run concurrently. Allowing multiple transactions to update data concurrently causes several complications with consistency of the data. For consistency, we can make transactions serially. However, there are two good reasons for allowing concurrency:

1. Improved throughput and resource utilization.
2. Reduced waiting time.

 Two actions are said to **conflict** if

1. They belong to **different transactions**. [one belong to T1 & other belongs to T2]
2. Both operations refers to the **same** database object.
3. One of them is **Write**.

T1	T2	CONFLICT (or) NOT
Read(A)	Read(A)	No
Read(A)	Write(A)	Conflict
Write(A)	Read(A)	Conflict
Write(A)	Write(A)	Conflict

Following are the problems encountered during concurrent transactions.

Dirty Read (WR conflict)

The dirty read problem occurs when one transaction updates an item of the database, and somehow the transaction fails, and before the data gets rollback, the updated database item is accessed by another transaction. There comes the Read-Write Conflict between both transactions.

For example:

Consider two transactions T_X and T_Y in the below diagram performing read/write operations on account A where the available balance in account A is \$300:

Time	T _x	T _y
t ₁	READ (A)	—
t ₂	A = A + 50	—
t ₃	WRITE (A)	—
t ₄	—	READ (A)
t ₅	SERVER DOWN ROLLBACK	—

DIRTY READ PROBLEM

- At time t₁, transaction T_x reads the value of account A, i.e., \$300.
- At time t₂, transaction T_x adds \$50 to account A that becomes \$350.
- At time t₃, transaction T_x writes the updated value in account A, i.e., \$350.
- Then at time t₄, transaction T_y reads account A that will be read as \$350.
- Then at time t₅, transaction T_x rollbacks due to server problem, and the value changes back to \$300 (as initially).
- But the value for account A remains \$350 for transaction T_y as committed, which is the dirty read and therefore known as the Dirty Read Problem.

Unrepeatable Read/Incorrect Summary (RW conflict):

Also known as Inconsistent Retrievals Problem that occurs when in a transaction, two different values are read for the same database item.

For example:

Consider two transactions, T_x and T_y, performing the read/write operations on account A, having an available balance = \$300. The diagram is shown below:

Time	T _x	T _y
t ₁	READ (A)	—
t ₂	—	READ (A)
t ₃	—	A = A + 100
t ₄	—	WRITE (A)
t ₅	READ (A)	—

UNREPEATABLE READ PROBLEM

- At time t₁, transaction T_x reads the value from account A, i.e., \$300.
- At time t₂, transaction T_y reads the value from account A, i.e., \$300.

- At time t3, transaction T_Y updates the value of account A by adding \$100 to the available balance, and then it becomes \$400.
- At time t4, transaction T_Y writes the updated value, i.e., \$400.
- After that, at time t5, transaction T_X reads the available value of account A, and that will be read as \$400.
- It means that within the same transaction T_X, it reads two different values of account A, i.e., \$ 300 initially, and after updation made by transaction T_Y, it reads \$400. It is an unrepeatable read and is therefore known as the Unrepeatable read problem.

Thus, in order to maintain consistency in the database and avoid such problems that take place in concurrent execution, management is needed, and that is where the concept of Concurrency Control comes into role.

Blind Write/ Lost Update (WW conflict):

The problem occurs when two different database transactions perform the read/write operations on the same database items in an interleaved manner (i.e., concurrent execution) that makes the values of the items incorrect hence making the database inconsistent.

For example:

Consider the below diagram where two transactions T_X and T_Y, are performed on the same account A where the balance of account A is \$300.

Time	T _x	T _y
t ₁	READ (A)	—
t ₂	A = A - 50	—
t ₃	—	READ (A)
t ₄	—	A = A + 100
t ₅	—	—
t ₆	WRITE (A)	—
t ₇	—	WRITE (A)

LOST UPDATE PROBLEM

- At time t1, transaction T_X reads the value of account A, i.e., \$300 (only read).
- At time t2, transaction T_X deducts \$50 from account A that becomes \$250 (only deducted and not updated/write).
- Alternately, at time t3, transaction T_Y reads the value of account A that will be \$300 only because T_X didn't update the value yet.
- At time t4, transaction T_Y adds \$100 to account A that becomes \$400 (only added but not updated/write).
- At time t6, transaction T_X writes the value of account A that will be updated as \$250 only, as T_Y didn't update the value yet.

- Similarly, at time t_7 , transaction T_Y writes the values of account A, so it will write as done at time t_4 that will be \$400. It means the value written by T_X is lost, i.e., \$250 is lost.

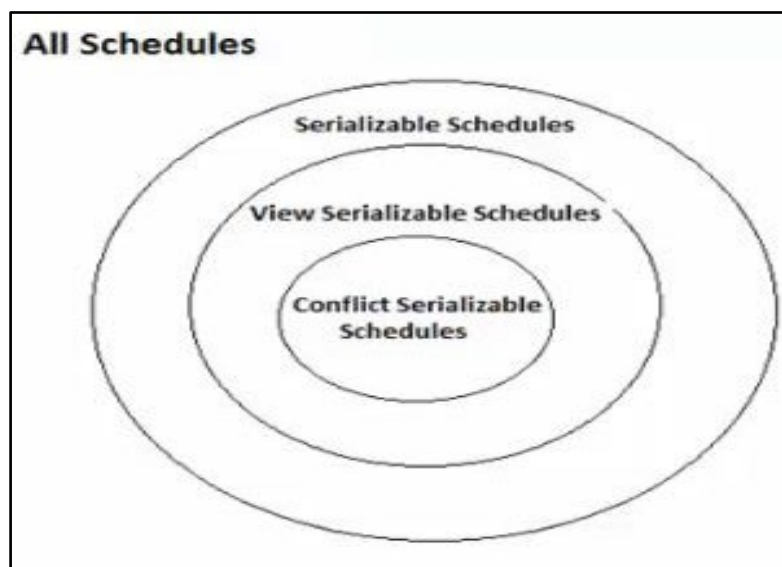
Hence data becomes incorrect, and database sets to inconsistent.

Some concurrency control mechanisms are **Locking, Timestamp, Multiple versions and Snapshot isolation**. Oracle uses **LOCKING** mechanism to control the concurrency.

Serializability

A schedule 'S' is said to be serializable if and only if the net effect is identical to some complete serial schedule over 'S'.

We use following types of serializability to verify Isolation property:



Conflict Serializability

We say that T_1 and T_2 **conflict** if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation.

If a schedule S can be transformed into a schedule S' by a series of swaps of nonconflicting instructions, we say that S and S' are **conflict equivalent**. We say that a schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule.

For determining conflict serializability of a schedule. Consider a schedule S . We construct a directed graph, called a **precedence graph**, from S . 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 participating in the schedule. The set of edges consists of all edges $T_i \rightarrow T_j$ for which one of three conditions holds:

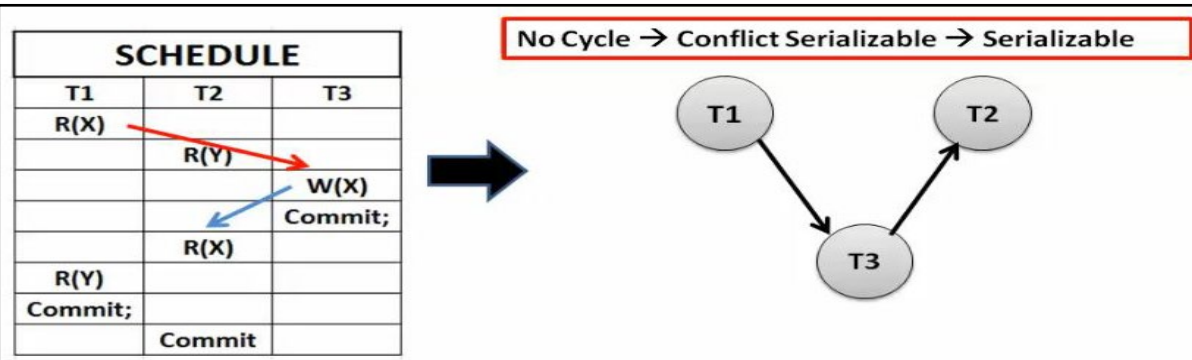
1. T_i executes write(Q) before T_j executes read(Q). ($W \rightarrow R$)
2. T_i executes read(Q) before T_j executes write(Q). ($R \rightarrow W$)
3. T_i executes write(Q) before T_j executes write(Q). ($W \rightarrow W$)

If the precedence graph for S has a cycle, then schedule S is not conflict serializable. If the graph contains no cycles (or it is acyclic), then the schedule S is conflict serializable.

A **serializability order** of the transactions can be obtained by finding a linear order consistent with the partial order of the precedence graph. This process is called **topological sorting**. There are, in general, several possible linear orders that can be obtained through a topological sort.

Steps to draw a Precedence graph:-

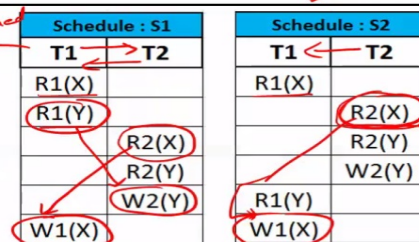
1. A node for each committed transaction in S .
2. An arc from T_i to T_j if an action of T_i precedes and conflicts with one of T_j 's actions.
3. A schedule ' S ' is Conflict Serializable, if and only if, the graph is **acyclic**.



Previous GATE Questions

Consider the following schedules involving the transactions. Which one of the following statements is TRUE ? $T_2 \rightarrow T_1$

$S_1 : r_1(X); r_1(Y); r_2(X); r_2(Y); w_2(Y); w_1(X)$
 $S_2 : r_1(X); r_2(X); r_2(Y); w_2(Y); r_1(Y); w_1(X)$



- Both S_1 and S_2 are conflict serializable
- S_1 is conflict serializable and S_2 is not conflict serializable
- S_1 is not conflict serializable and S_2 is conflict serializable
- Both S_1 and S_2 are not conflict serializable

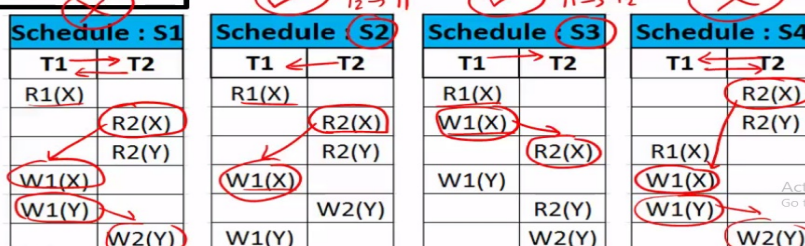
2

Which of the following schedules are **conflict serializable** ?

$T_1 : R_1[x] W_1[x] W_1[y]$
 $T_2 : R_2[x] R_2[y] W_2[y]$

$S_1 : R_1[x] R_2[x] R_2[y] W_1[x] W_1[y] W_2[y]$
 $S_2 : R_1[x] R_2[x] R_2[y] W_1[x] W_2[y] W_1[y]$
 $S_3 : R_1[x] W_1[x] R_2[x] W_1[y] R_2[y] W_2[y]$
 $S_4 : R_2[x] R_2[y] R_1[x] W_1[x] W_1[y] W_2[y]$

- S_1 and S_2
- S_2 and S_3
- S_3 only
- S_4 only



3

Every conflict serializable schedule is serializable. If a serial isn't conflict serializable, it may or may not be serializable.

View Serializability

Two schedules S_1 and S_2 are said to be view equivalent if the following three conditions hold:

1. If transaction T_i reads the initial value of data item (x) in schedule S_1 , then transaction T_i must read the initial value of (x) in schedule S_2 .
2. In schedule S_1 , if transaction T_i reads the value of data item (x) produced by transaction T_j then T_i must reads the value of (x) that produced by transaction T_j in the schedule S_2 .
3. If transaction T_i write the final value of data item (x) in schedule S_1 , then transaction T_i must write the final value of (x) in schedule S_2 .

A schedule will view serializable if it is view equivalent to a serial schedule. If a schedule is conflict serializable, then it will be view serializable. The view serializable which does not conflict serializable contains blind writes.

View Equivalent

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

1. Initial Reads

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.

T1	T2		T1	T2
Read(A)	Write(A)		Read(A)	Write(A)
Schedule S1			Schedule S2	

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 or W-R Conflicts

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 .

T1	T2	T3	T1	T2	T3
Write(A)	Write(A)	Read(A)	Write(A)	Write(A)	<u>Read(A)</u>
Schedule S1			Schedule S2		

Above two schedules are not view equal because, in S_1 , T_3 is reading A updated by T_2 and 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 writes operations should also be done by T_1 .

T1	T2	T3	T1	T2	T3
Write(A)	Read(A)	Write(A)	Write(A)	Read(A)	Write(A)
Schedule S1			Schedule S2		

Above two schedules are 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 .

Taking first schedule S_1 :

T1	T2	T3
Read(A) Write(A)	Write(A)	Write(A)

Schedule S_1

Step 1: Final updation on 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 and in S_1 , it is also done by T_3 . So, S and S_1 are view Equivalent.

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

Database Object	Initial Read	Write-Read Sequence	Final Write
A	T_1	-	T_3

Hence, view equivalent serial schedule is:

$T_1 \rightarrow T_2 \rightarrow T_3$

→ Verify whether the given Schedule is Serializable (or) not?

T1 → Transfers 10 rupees from X to Y Is Schedule Conflict Serializable ? **NO**

T2 → Transfers 20 rupees from X to Y Is Schedule View Serializable ? **NO**

T1	T2
R(X)	
X=X-10	
W(X)	
	R(Y)
	Y=Y+20
	W(Y)
R(Y)	
Y=Y+10	
W(Y)	
COMMIT;	
	R(X)
	X=X-20
	W(X)
	COMMIT;

TEST FOR CONFLICT SERIALIZABILITY :-

Cycle is formed → Not Conflict Serializable

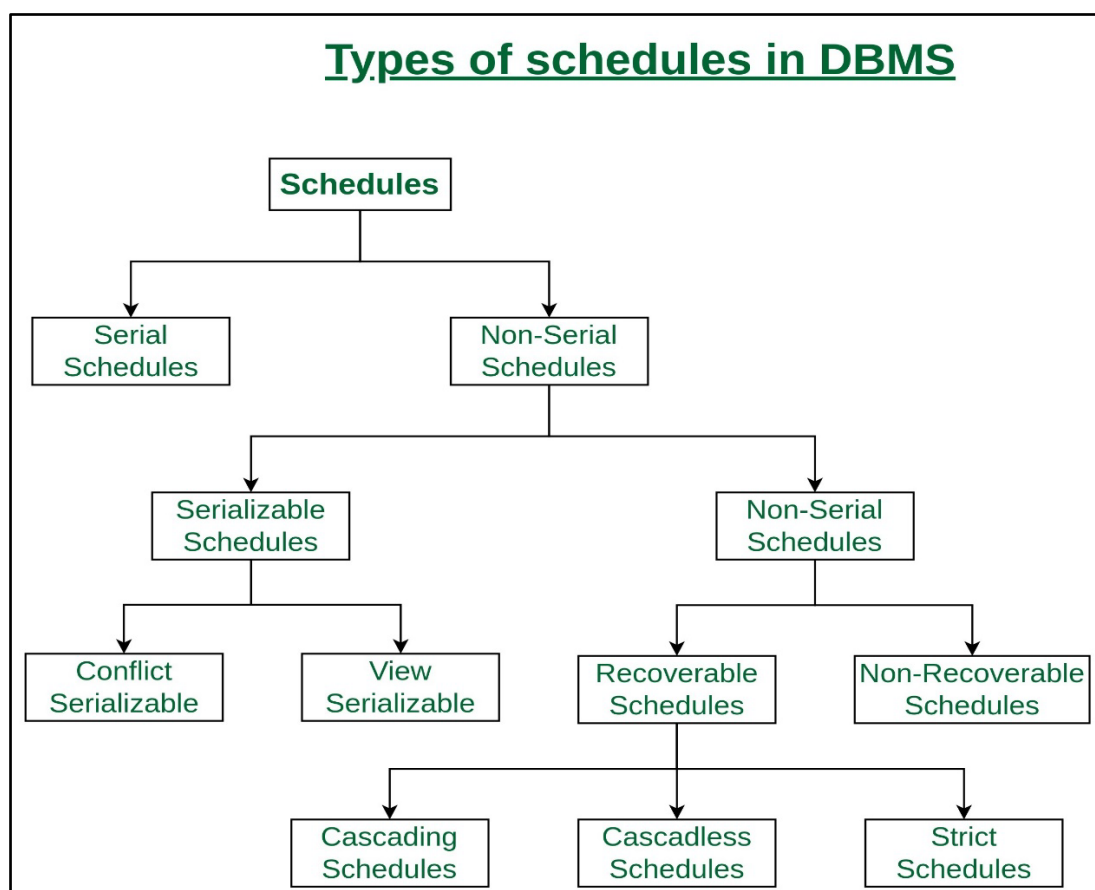
TEST FOR VIEW SERIALIZABILITY :-

SNO	OBJECT	INITIAL READ	WRITE READ SEQUENCE	FINAL WRITE
1	X	T1	T1 → T2	T2
2	Y	T2	T2 → T1	T1

Serial Order from OBJECT X = T1 → T2
Serial Order from OBJECT Y = T2 → T1

Impossible
Schedule is Not VIEW SERIALIZABLE

Types of Schedules



Serial Schedules: Schedules in which the transactions are executed non-interleaved, i.e., a serial schedule is one in which no transaction starts until a running transaction has ended are called serial schedules.

T ₁	T ₂
R(A)	
W(A)	
R(B)	
	W(B)
	R(A)
	R(B)

Non-Serializable:

The non-serializable schedule is divided into two types, Recoverable and Non-recoverable Schedule.

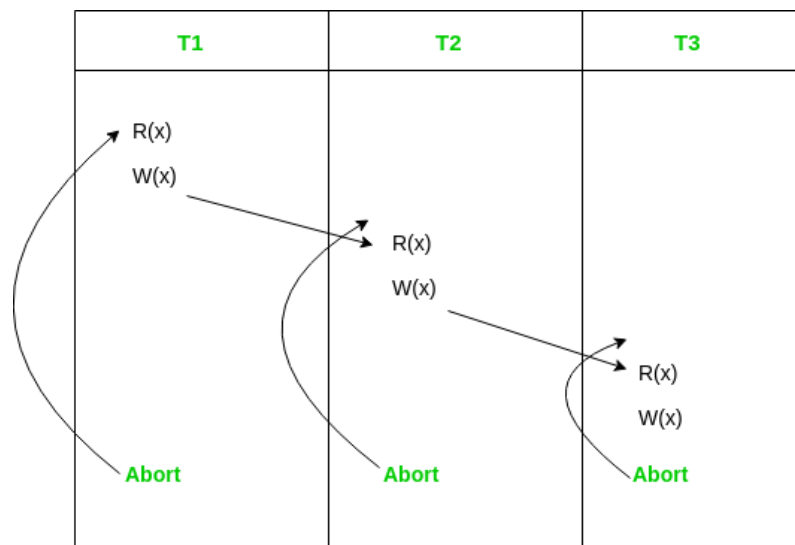
1. Recoverable Schedule: Schedules in which transactions commit only after all transactions whose changes they read commit are called recoverable schedules. In other words, if some transaction T_j is reading value updated or written by some other transaction T_i, then the commit of T_j must occur after the commit of T_i.

T ₁	T ₂
R(A)	
W(A)	
	W(A)
	R(A)
commit	
	commit

2. Non-recoverable Schedule:

T ₁	T ₂
R(A)	
W(A)	
	W(A)
	R(A)
	commit
abort	

Cascading Schedule: Also called Avoids cascading aborts/rollbacks (ACA). When there is a failure in one transaction and this leads to the rolling back or aborting other dependent transactions, then such scheduling is referred to as Cascading rollback or cascading abort.



Cascade less Schedule: Schedules in which transactions read values only after all transactions whose changes they are going to read commit are called cascade less schedules. Avoids that a single transaction abort leads to a series of transaction rollbacks. A strategy to prevent cascading aborts is to disallow a transaction from reading uncommitted changes from another transaction in the same schedule.

T ₁	T ₂
R(A)	
W(A)	
	W(A)
commit	
	R(A)
	commit

This schedule is cascade less. Since the updated value of **A** is read by T₂ only after the updating transaction i.e., T₁ commits.

Strict Schedule:

A schedule is strict if for any two transactions T_i, T_j, if a write operation of T_i precedes a conflicting operation of T_j (either read or write), then the commit or abort event of T_i also precedes that conflicting operation of T_j.

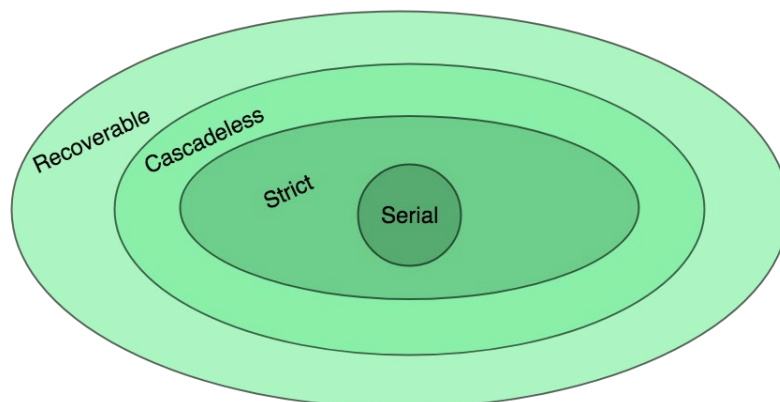
The following schedule is a strict schedule since T₂ reads and writes A which is written by T₁ only after the commit of T₁.

T ₁	T ₂
R(A)	
	R(A)
W(A)	
commit	
	W(A)
	R(A)
	commit

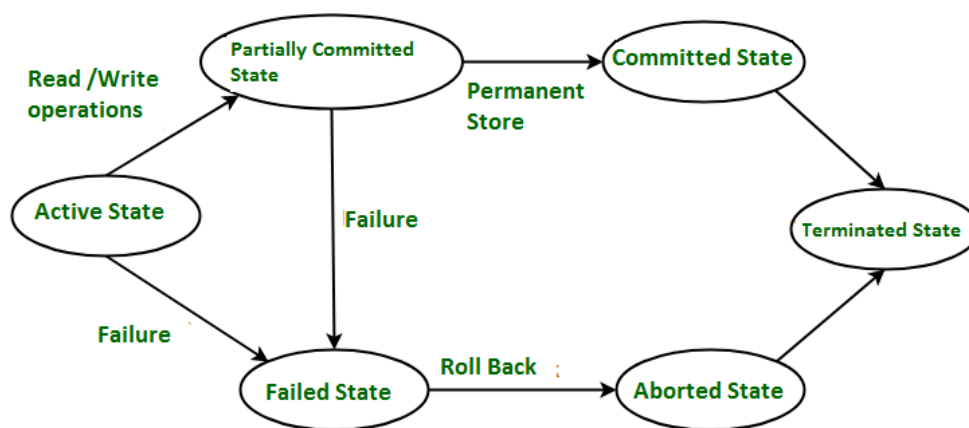
It can be seen that:

1. Cascade less schedules are stricter than recoverable schedules or are a subset of recoverable schedules.
2. Strict schedules are stricter than cascade less schedules or are a subset of cascade less schedules.
3. Serial schedules satisfy constraints of all recoverable, cascade less and strict schedules and hence is a subset of strict schedules.

The relation between various types of schedules can be depicted as:



Transaction States



Transaction States in DBMS

These are the states which tell about the current state of the Transaction and also tell how we will further do processing we will do on the transactions. These states govern the rules which decide the fate of the transaction whether it will commit or abort.

Active State – When the instructions of the transaction is running then the transaction is in active state. If all the read and write operations are performed without any error then it goes to “partially committed state”, if any instruction fails it goes to “failed state”.

Partially Committed – After completion of all the read and write operation the changes are made in main memory or local buffer. If the changes are made permanent on the Data Base then state will change to “committed state” and in case of failure it will go to “failed state”.

Failed State – When any instruction of the transaction fails it goes to “failed state” or if failure occurs in making permanent change of data on Database.

Aborted State – After having any type of failure the transaction goes from “failed state” to “aborted state” and in before states the changes are only made to local buffer or main memory and hence these changes are deleted or rollback.

Committed Stage – It is the stage when the changes are made permanent on the Data Base and transaction is complete and therefore terminated in “terminated state”.

Terminated State – If there is any roll back or the transaction come from “committed state” then the system is consistent and ready for new transaction and the old transaction is terminated.

Lock Management

One way to ensure isolation is to require that data items be accessed in a mutually exclusive manner; that is, while one transaction is accessing a data item, no other transaction can modify that data item. The most common method used to implement this requirement is to allow a transaction to access a data item only if it is currently holding a **lock** on that item.

The part of DBMS that keeps track of the lock issued to a transaction is called **Lock Manager**. The lock manager maintains an entry of each object identifier in a **lock table**. DBMS maintains an entry of each transaction in **transaction table**.

A **lock manager** can be implemented as a process that receives messages from transactions and sends messages in reply. The lock-manager process replies to **lock-request** messages with **lock-grant** messages, or with messages requesting rollback of the transaction (in case of deadlocks). **Unlock** messages require only an acknowledgment in response but may result in a grant message to another waiting transaction.

Binary Lock:

A **binary lock** can have two **states** or **values**: locked and unlocked (or 1 and 0, for simplicity). A distinct lock is associated with each database item X . If the value of the lock on X is 1, item X *cannot be accessed* by a database operation that requests the item. If the value of the lock on X is 0, the item can be accessed when requested, and the lock value is changed to 1. We refer to the current value (or state) of the lock associated with item X as **lock(X)**. A binary lock enforces **mutual exclusion** on the data item.

The `lock_item` and `unlock_item` operations must be implemented as indivisible units (known as **critical sections** in operating systems); that is, no interleaving should be allowed once a lock or unlock operation is started until the operation terminates or the transaction waits.

If the simple binary locking scheme described here is used, every transaction must obey the following rules:

1. A transaction T must issue the operation `lock_item(X)` before any `read_item(X)` or `write_item(X)` operations are performed in T .
2. A transaction T must issue the operation `unlock_item(X)` after all `read_item(X)` and `write_item(X)` operations are completed in T .
3. A transaction T will not issue a `lock_item(X)` operation if it already holds the lock on item X .
4. A transaction T will not issue an `unlock_item(X)` operation unless it already holds the lock on item X .

Shared/Exclusive Lock (Read/Write Lock):

1. **Shared.** If a transaction T_i has obtained a **shared-mode lock** (denoted by S) on item Q , then T_i can read, but cannot write, Q .
2. **Exclusive.** If a transaction T_i has obtained an **exclusive-mode lock** (denoted by X) on item Q , then T_i can both read and write Q .

We require that every transaction **request** a lock in an appropriate mode on data item Q , depending on the types of operations that it will perform on Q . The transaction makes the request to the concurrency-control manager. The transaction can proceed with the operation only after the concurrency-control manager **grants** the lock to the transaction. The use of these two lock modes allows multiple transactions to read a data item but limits write access to just one transaction at a time.

Lock-compatibility Matrix		
	Shared	Exclusive
Shared	T	F
Exclusive	F	F

given a set of lock modes, we can define a **compatibility function** on them as follows: Let A and B represent arbitrary lock modes. Suppose that a transaction T_i requests a lock of mode A on item Q on which transaction T_j ($T_i \neq T_j$) currently holds a lock of mode B . If transaction T_i can be granted a lock on Q immediately, in spite of the presence of the mode B lock, then we say mode A is **compatible** with mode B . Such a function can be represented conveniently by a matrix.

A lock associated with an item X , `LOCK(X)`, now has three possible states: *read-locked*, *write-locked*, or *unlocked*. A **read-locked item** is also called **share-locked** because other transactions are allowed to read the item, whereas a **write-locked item** is called **exclusive-locked** because a single transaction exclusively holds the lock on the item.

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, the concurrency control manager will not grant the lock until all incompatible locks held by other transactions have been released. Thus, T_i is made to **wait** until all incompatible locks held by other transactions have been released.

Transaction T_i may unlock a data item that it had locked at some earlier point. Note that a transaction must hold a lock on a data item as long as it accesses that item. Moreover, it is not necessarily desirable for a transaction to unlock a data item immediately after its final access of that data item, since serializability may not be ensured.

The transaction making a lock request cannot execute its next action until the concurrency control manager grants the lock. Hence, the lock must be granted in the interval of time between the lock-request operation and the following action of the transaction.

If we do not use locking, or if we unlock data items too soon after reading or writing them, we may get inconsistent states. On the other hand, if we do not unlock a data item before requesting a lock on another data item, deadlocks may occur.

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. Locking protocols restrict the number of possible schedules. The set of all such schedules is a proper subset of all possible serializable schedules. We shall present several locking protocols that allow only conflict-serializable schedules, and thereby ensure isolation.

Let $\{T_0, T_1, \dots, T_n\}$ be a set of transactions participating in a schedule S . We say that T_i **precedes** T_j in S , written $T_i \rightarrow T_j$, if there exists a data item Q such that T_i has held lock mode A on Q , and T_j has held lock mode B on Q later, and $\text{comp}(A, B) = \text{false}$. If $T_i \rightarrow T_j$, then that precedence implies that in any equivalent serial schedule, T_i must appear before T_j . Conflicts between instructions correspond to non-compatibility of lock modes.

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; in other words, for all legal schedules the associated \rightarrow relation is acyclic.

Granting of Locks

Problem in shared/exclusive locking:

1. May not free from irrecoverability
2. May not be free from deadlock
3. May not be free from starvation

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 never gets the exclusive-mode lock on the data item. The transaction T 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:

1. There is no other transaction holding a lock on Q in a mode that conflicts with M .
2. There is no other transaction that is waiting for a lock on Q and that made its lock request before T_i .

Locking Protocols

Two-Phase Locking(2PL) Protocol

One protocol that ensures serializability is the **two-phase locking protocol**. This protocol requires that each transaction issue lock and unlock requests in two phases:

1. **Growing/Expanding phase.** A transaction may obtain locks, but may not release any lock.
2. **Shrinking phase.** A transaction may release locks, but may not obtain any new locks.

We denote conversion from shared to exclusive modes by **upgrade**, and from exclusive to shared by **downgrade**. Lock conversion cannot be allowed arbitrarily. If lock conversion is allowed, then upgrading of locks (from read-locked to write-locked) must be done during the expanding phase, and downgrading of locks (from write-locked to read-locked) must be done in the shrinking phase.

Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests.

We can show that the two-phase locking protocol ensures conflict serializability. Consider any transaction. The point in the schedule where the transaction has obtained its final lock (the end of its growing phase) is called the **lock point** of the transaction. Now, transactions can be ordered according to their lock points - this ordering is, in fact, a serializability ordering for the transactions.

Advantage: always ensures serializability

Drawback:

1. May not free from irrecoverability
2. May not be free from deadlock
3. May not be free from starvation
4. May not be free from cascading rollback

Strict Two-Phase Locking (Strict-2PL) Protocol

It is most popular protocol in practice. Cascading rollbacks can be avoided by a modification of two-phase locking called the **strict two-phase locking protocol**. This protocol requires not only that locking be two phases, but also 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.

Rigorous Two-Phase Locking (Rigorous-2PL) Protocol

Another variant of two-phase locking is the **rigorous two-phase locking protocol**, which requires that all locks be held until the transaction commits. It is more restrictive variation of strict-2PL. We can easily verify that, with rigorous two-phase locking, transactions can be serialized in the order in which they commit.

Conservative/Static Two-Phase Locking (Static-2PL) Protocol

A variation known as **conservative 2PL** (or **static 2PL**) requires a transaction to lock all the items it accesses *before the transaction begins execution*, by **predeclaring** its *read-set* and

write-set. The **read-set** of a transaction is the set of all items that the transaction reads, and the **write-set** is the set of all items that it writes. If any of the predeclared items needed cannot be locked, the transaction does not lock any item; instead, it waits until all the items are available for locking. Conservative 2PL is a **deadlock-free protocol**. However, it is difficult to use in practice because of the need to predeclare the read-set and write-set, which is not possible in some situations. It is just a theory of Machine Learning in current time to get free from deadlocks.

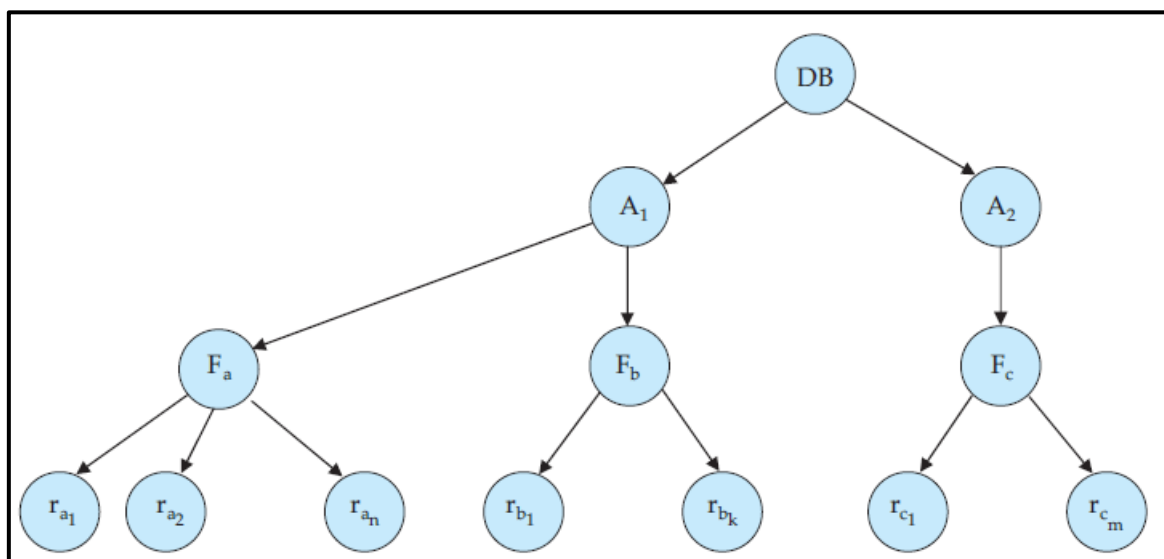
Multiple Granularity Locking (MGL) Protocol

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 five levels of nodes. The highest level represents the entire database. Below there 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. A file collection of *pages*. Finally, each page has nodes of type *record*. As before, the page consists of exactly those records that are its child nodes, and no record can be present in more than one page.



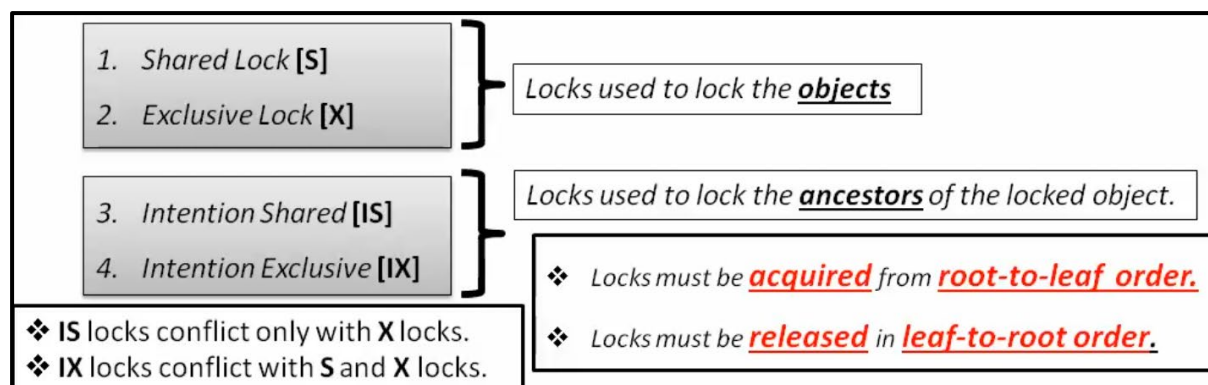
Each node in the tree can be locked individually. As we did in the two-phase locking protocol, we shall use **shared** and **exclusive** lock modes. When a transaction locks a node, in either shared or exclusive mode, the transaction also has implicitly locked all the descendants of that node in the 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.

Suppose that transaction T_j wishes to lock record r_{b6} of file F_b . Since T_i has locked F_b explicitly, it follows that r_{b6} is also locked (implicitly). But, when T_j issues a lock request for r_{b6} , r_{b6} is not explicitly locked! How does the system determine whether T_j can lock r_{b6} ? T_j must traverse the tree from the root to record r_{b6} . If any node in that path is locked in an incompatible mode, then T_j must be delayed.

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.

Compatibility Matrix for lock modes					
	IS	IX	S	SIX	X
IS	T	T	T	T	F
IX	T	T	F	F	F
S	T	F	T	F	F
SIX	T	F	F	F	F
X	F	F	F	F	F



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.
2. Transaction T_i must lock the root of the tree first, and can lock it in anymode.
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.

The multiple granularity level protocol is especially suited when processing a mix of transactions that include (1) short transactions that access only a few items (records or fields) and (2) long transactions that access entire files. In this environment, less transaction blocking and less locking overhead are incurred by such a protocol when compared to a single-level granularity locking approach.

Deadlock is possible in the multiple-granularity protocol, as it is in the two-phase locking protocol. There are techniques to reduce deadlock frequency in the multiple-granularity protocol, and also to eliminate deadlock entirely.

Two-Phase Commit (2PC) Protocol

To maintain the atomicity of a multi-database transaction, it is necessary to have a two-level recovery mechanism. A **global recovery manager**, or **coordinator**, is needed to maintain information needed for recovery, in addition to the local recovery managers and the information they maintain (log, tables). The coordinator usually follows a protocol called the **two-phase commit protocol**; whose two phases can be stated as follows:

- **Phase 1. (Prepare phase)**
 - After each slave has locally completed its transaction, it sends a “DONE” message to the controlling site. When the controlling site has received “DONE” message from all slaves, it sends a “Prepare” message to the slaves.
 - The slaves vote on whether they still want to commit or not. If a slave wants to commit, it sends a “Ready” message.
 - A slave that does not want to commit sends a “Not Ready” message. This may happen when the slave has conflicting concurrent transactions or there is a timeout.
- **Phase 2 (Commit/Abort phase)**
 - After the controlling site has received “Ready” message from all the slaves –
 - The controlling site sends a “Global Commit” message to the slaves.
 - The slaves apply the transaction and send a “Commit ACK” message to the controlling site.
 - When the controlling site receives “Commit ACK” message from all the slaves, it considers the transaction as committed.

- After the controlling site has received the first “Not Ready” message from any slave –
 - The controlling site sends a “Global Abort” message to the slaves.
 - The slaves abort the transaction and send a “Abort ACK” message to the controlling site.
 - When the controlling site receives “Abort ACK” message from all the slaves, it considers the transaction as aborted.

The net effect of the two-phase commit protocol is that either all participating databases commit the effect of the transaction or none of them do. In case any of the participants or the coordinator fails, it is always possible to recover to a state where either the transaction is committed or it is rolled back. A failure during or before phase 1 usually requires the transaction to be rolled back, whereas a failure during phase 2 means that a successful transaction can recover and commit.

Timestamp ordering Protocol

A **timestamp** is a unique identifier created by the DBMS to identify a transaction. Typically, timestamp values are assigned in the order in which the transactions are submitted to the system, so a timestamp can be thought of as the *transaction start time*. We will refer to the timestamp of transaction T as $TS(T)$. Concurrency control techniques based on timestamp ordering do not use locks; hence, *deadlocks cannot occur*.

This timestamp is assigned by the database system before the transaction T_i starts execution. If a transaction T_i has been assigned timestamp $TS(T_i)$, and a new transaction T_j enters the system, then $TS(T_i) < TS(T_j)$.

If timestamp isn't assigned, then whichever transaction performs operation earlier is considered as older than other transactions.

There are two simple methods for implementing this scheme:

1. Use the value of the **system clock** as the timestamp; that is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system.
2. Use a **logical counter** that is incremented after a new timestamp has been assigned; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system.

The timestamps of the transactions determine 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 . Here, T_i is called older transaction as it came first and T_j is called younger transaction as it came later.

To implement this scheme, we associate with each data item Q two timestamp values:

- **W-timestamp**(Q) denotes the largest timestamp of any transaction that executed $write(Q)$ successfully.
- **R-timestamp**(Q) denotes the largest timestamp of any transaction that executed $read(Q)$ successfully.

The **timestamp-ordering protocol** ensures that any conflicting read and write operations are executed in timestamp order. This protocol operates as follows:

1. Suppose that transaction T_i issues $read(Q)$.

- a. If $TS(T_i) < W\text{-timestamp}(Q)$, then T_i needs to read a value of Q that was already overwritten. Hence, the read operation is rejected, and T_i is rolled back.
- b. If $TS(T_i) \geq W\text{-timestamp}(Q)$, then the read operation is executed, and $R\text{-timestamp}(Q)$ is set to the maximum of $R\text{-timestamp}(Q)$ and $TS(T_i)$.
2. Suppose that transaction T_i issues $write(Q)$.
 - a. If $TS(T_i) < R\text{-timestamp}(Q)$, then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced. Hence, the system rejects the write operation and rolls T_i back.
 - b. If $TS(T_i) < W\text{-timestamp}(Q)$, then T_i is attempting to write an obsolete value of Q . Hence, the system rejects this write operation and rolls T_i back.
 - c. Otherwise, the system executes the write operation and sets $W\text{-timestamp}(Q)$ to $TS(T_i)$.

If a transaction T_i is rolled back by the concurrency-control scheme as result of issuance of either a read or write operation, the system assigns it a new timestamp and restarts it.

Shortcut Method to check validity of Schedule as per Timestamp ordering Protocol:

If there exist a conflict in schedule and also there exist a YO sequence (A conflict operation in such an order that the operation is performed in Younger transaction first and then in older transaction), then schedule is invalid.

Conflict	YO sequence	Schedule
T	T	Not Valid
T	F	Valid
F	T	Valid
F	F	Valid

Deadlock Handling

A system is in a deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set. More precisely, there exists a set of waiting transactions $\{T_0, T_1, \dots, T_n\}$ such that T_0 is waiting for a data item that T_1 holds, and T_1 is waiting for a data item that T_2 holds, and ... and T_{n-1} is waiting for a data item that T_n holds, and T_n is waiting for a data item that T_0 holds. None of the transactions can make progress in such a situation.

The only remedy to this undesirable situation is for the system to invoke some drastic action, such as rolling back some of the transactions involved in the deadlock. Rollback of a transaction may be partial: That is, a transaction may be rolled back to the point where it obtained a lock whose release resolves the deadlock.

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. 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. As we shall see, both methods may result in transaction rollback. Prevention is commonly used if the probability that the system would enter a deadlock state is relatively high; otherwise, detection and recovery are more efficient.

Deadlock Prevention

There are two approaches to deadlock prevention. One approach ensures that no cyclic waits can occur by ordering the requests for locks, or requiring all locks to be acquired together. 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.

The second approach for preventing deadlocks is to use preemption and transaction rollbacks. In preemption, 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, based on a counter or on the system clock, 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:

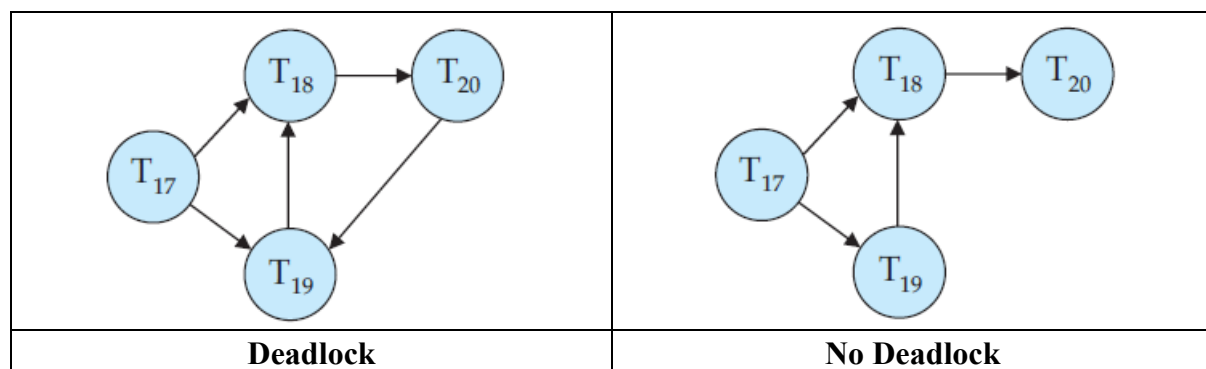
1. The **wait-die** scheme is a non-preemptive technique. When transaction T_i requests a data item currently held by T_j , T_i is allowed to wait only if it has a timestamp smaller than that of T_j (that is, T_i is older than T_j). Otherwise, T_i is rolled back (dies).
2. The **wound-wait** scheme is 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).

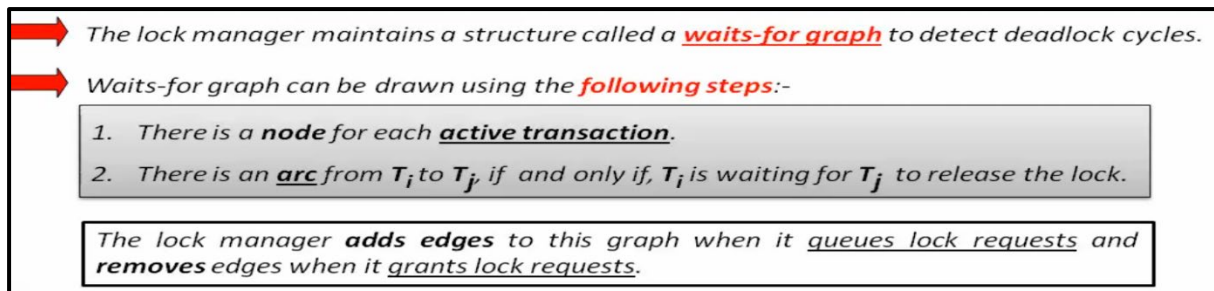
Deadlock Detection

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.

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

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.





Starvation

Another problem that may occur when we use locking is **starvation**, which occurs when a transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally. This may occur if the waiting scheme for locked items is unfair in that it gives priority to some transactions over others. One solution for starvation is to have a fair waiting scheme, such as using a **first-come-first-served** queue; transactions are enabled to lock an item in the order in which they originally requested the lock. Another scheme allows some transactions to have priority over others but increases the priority of a transaction the longer it waits, until it eventually gets the highest priority and proceeds. Starvation can also occur because of victim selection if the algorithm selects the same transaction as victim repeatedly, thus causing it to abort and never finish execution. The algorithm can use higher priorities for transactions that have been aborted multiple times to avoid this problem. The wait-die and wound-wait schemes discussed previously avoid starvation, because they restart a transaction that has been aborted with its same original timestamp, so the possibility that the same transaction is aborted repeatedly is slim.

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:

1. **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:
 - a. How long the transaction has computed, and how much longer the transaction will compute before it completes its designated task.
 - b. How many data items the transaction has been used?
 - c. How many more data items the transaction needs for it to complete.
 - d. How many transactions will be involved in the rollback?
2. **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.

3. **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.

Log based Recovery

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

There are several types of log records. An **update log record** describes a single database write. It has these fields:

- **Transaction identifier**, which is the unique identifier of the transaction that performed the write operation.
- **Data-item identifier**, which is the unique identifier of the data item written. Typically, it is the location on disk of the data item, consisting of the block identifier of the block on which the data item resides, and an offset within the block.
- **Old value**, which is the value of the data item prior to the write.
- **New value**, which is the value that the data item will have after the write.

Other special log records exist to record significant events during transaction processing, such as the start of a transaction and the commit or abort of a transaction. Among the types of log records are:

- $\langle T_i \text{ start} \rangle$. Transaction T_i has started.
- $\langle T_i \text{ commit} \rangle$. Transaction T_i has committed.
- $\langle T_i \text{ abort} \rangle$. Transaction T_i has aborted.

We say a transaction *modifies the database* if it performs an update on a disk buffer, or on the disk itself; updates to the private part of main memory do not count as database modifications. If a transaction does not modify the database until it has committed, it is said to use the **deferred-modification** technique. If database modifications occur while the transaction is still active, the transaction is said to use the **immediate-modification** technique. Deferred modification has the overhead that transactions need to make local copies of all updated data items; further, if a transaction reads a data item that it has updated, it must read the value from its local copy.

Deferred modification Log:

$\langle \text{TRANSACTION NUMBER, OBJECT, NEW VALUE} \rangle$

Immediate modification Log:

$\langle \text{TRANSACTION NUMBER, OBJECT, OLD VALUE, NEW VALUE} \rangle$

A recovery algorithm must take into account a variety of factors, including:

- The possibility that a transaction may have committed although some of its database modifications exist only in the disk buffer in main memory and not in the database on disk.
- The possibility that a transaction may have modified the database while in the active state and, as a result of a subsequent failure, may need to abort.

Because all database modifications must be preceded by the creation of a log record, the system has available both the old value prior to the modification of the data item and the new value that is to be written for the data item. This allows the system to perform *undo* and *redo* operations as appropriate.

- **Undo** using a log record sets the data item specified in the log record to the old value.
- **Redo** using a log record sets the data item specified in the log record to the new value.

We say that a transaction has **committed** when its commit log record, which is the last log record of the transaction, has been output to stable storage; at that point all earlier log records have already been output to stable storage. Thus, there is enough information in the log to ensure that even if there is a system crash, the updates of the transaction can be redone. If a system crash occurs before a log record $\langle T_i \text{ commit} \rangle$ is output to stable storage, transaction T_i will be rolled back. Thus, the output of the block containing the commit log record is the single atomic action that results in a transaction getting committed.

A = 100 B = 200 C = 500	<table><tr><th>Log</th></tr><tr><td><T1,START></td></tr><tr><td><T1,A,200></td></tr><tr><td><T1,B,400></td></tr><tr><td><T1,COMMIT></td></tr><tr><td><T2,START></td></tr><tr><td><T2,C,500></td></tr><tr><td><div><div>*</div><div>Failure</div></div></td></tr></table>	Log	<T1,START>	<T1,A,200>	<T1,B,400>	<T1,COMMIT>	<T2,START>	<T2,C,500>	<div><div>*</div><div>Failure</div></div>
Log									
<T1,START>									
<T1,A,200>									
<T1,B,400>									
<T1,COMMIT>									
<T2,START>									
<T2,C,500>									
<div><div>*</div><div>Failure</div></div>									
A = 200 B = 400 C = 500									

T1 WILL BE REDO, FOR T2 NO OPERATION WILL BE DONE AS IT IS NOT COMMITTED

A = 100 B = 200 C = 500	<table><tr><th>Log</th></tr><tr><td><T1,START></td></tr><tr><td><T1,A,100,200></td></tr><tr><td><T1,B,200,400></td></tr><tr><td><T1,COMMIT></td></tr><tr><td><T2,START></td></tr><tr><td><T2,C,500,600></td></tr><tr><td>*</td></tr></table>	Log	<T1,START>	<T1,A,100,200>	<T1,B,200,400>	<T1,COMMIT>	<T2,START>	<T2,C,500,600>	*
Log									
<T1,START>									
<T1,A,100,200>									
<T1,B,200,400>									
<T1,COMMIT>									
<T2,START>									
<T2,C,500,600>									
*									
A = 200 B = 400 C = 500									

T1 WILL BE REDO, T2 WILL BE UNDO