UNIT - IV
Introduction to Transactions

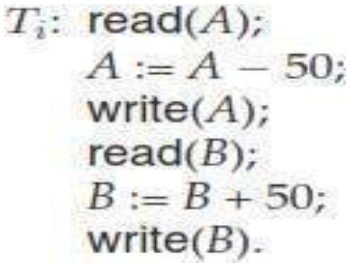
THE CONCEPT OF A TRANSACTION:

A transaction is a unit of program execution that accesses and possibly updates various data items. Usually, a transaction is initiated by a user program written in a high-level data-manipulation language or programming language.

Transactions access data using two operations:

- ***read(X),*** which transfers the data item X from the database to a local buffer belonging to the transaction that executed the read operation.
- ***write(X)***, which transfers the data item X from the local buffer of the transaction that executed the write back to the database.

Let Ti be a transaction that transfers $50 from account A to account B. This transaction can be defined as:



ACID Properties:

The acronym ACID is sometimes used to refer to the four properties of transactions that we have presented here: atomicity, consistency, isolation and durability. These ensure tomaintain data in the face of concurrent access and system failures:

Atomicity: Suppose that, just before the execution of transaction Ti the values of accounts A and B are $1000 and $2000, respectively. Now suppose that, during the execution of transaction Ti, a failure occurs that prevents Ti from completing its execution successfully. Examples of such failures include power failures, hardware failures, and software errors. 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 Ti 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, and, if the transaction does not complete its execution, the database system restores the old values to make it appear as though the transaction never executed.

Ensuring atomicity is the responsibility of the database system itself; specifically, it is handled by a component called the transaction-management component.

Consistency: The consistency requirement here is that the sum of A and B be unchanged by the execution of the transaction. 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.

Isolation: 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.

Ex: 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, as they allow multiple transactions to execute concurrently. The isolation property of a transaction ensures that the concurrent execution of transactions results in a system state that is equivalent to a state that could have been obtained had these transactions executed one at a time in some order. Ensuring the isolation property is the responsibility of a component of the database system called the concurrency-control component.

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 will 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 after the transaction completes execution. We assume for now that a failure of the computer system may result in loss of data in main memory, but data written to disk are never lost. We can guarantee durability by ensuring that either

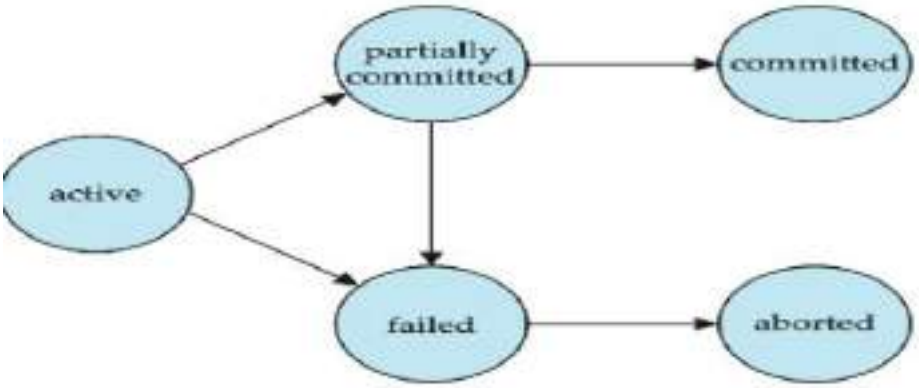
1. The updates carried out by the transaction have been written to disk before the transaction completes.

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.

Ensuring durability is the responsibility of a component of the database system called the recovery-management component.

2. TransactionState

Transaction State Diagram: A simple abstract transaction model is shown in fig below:



A transaction must be in one of the following states:

- *Active*, the initial state; the transaction stays in this state while it is executing
- *Partially committed*, after the final statement has been executed
- *Failed*, after the discovery that normal execution can no longer proceed
- *Aborted*, after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction
- *Committed*, after successful completion.

A transaction starts in the active state. When it finishes its final statement, it enters the partially committed state. At this point, the transaction has completed its execution, but it is still possible that it may have to be aborted, since the actual output may still be temporarily residing in main memory, and thus a hardware failure may preclude its successful completion.

The database system then writes out enough information to disk that, even in the event of a failure, the updates performed by the transaction can be re-created when the system restarts after the failure. When the last of this information is written out, the transaction enters the committed state. As mentioned earlier, we assume for now that failures do not result in loss of data on disk.

A transaction enters the failed state after the system determines that the transaction can no longer proceed with its normal execution (for example, because of hardware or logical errors). Such a transaction must be rolled back. Then, it enters the aborted state. At this point, the system has two options:

→It can restart the transaction, but only if the transaction was aborted as a result of some hardware or software error that was not created through the internal logic of the transaction. A restarted transaction is considered to be a new transaction.

→It can kill the transaction. It usually does so because of some internal logical error that can be corrected only by rewriting the application program, or because the input was bad, or because the desired data were not found in the database.

3. Implementation of Atomicity and Durability

- The recovery-management component of a database system can support atomicity and durability by a variety of schemes. We first consider a simple, but extremely inefficient, scheme called the *shadow copy scheme.* In the shadow-copy scheme, a transaction that wants to update the database first creates a complete copy of the database. All updates are done on the new database copy, leaving the original copy, the shadow copy, untouched. If at any point the transaction has to be aborted, the system merely deletes the new copy. The old copy of the database has not been affected.
- This scheme is based on making copies of the database, called shadow copies, assumes that only one transaction is active at a time. The scheme also assumes that the database is simply a file on disk. A pointer called db-pointer is maintained on disk; it points to the current copy of the database.

If the transaction completes, it is committed as follows:

- First, the operating system is asked to make sure that all pages of the new copy of the database have been written out to disk. (Unix systems use the flush command for this purpose.)
- After the operating system has written all the pages to disk, the database system updates the pointer db-pointer to point to the new copy of the database; the new copy then becomes the current copy of the database. The old copy of the database is then deleted.

The transaction is said to have been committed at the point where the updated dbpointer is written to disk.

We now consider how the technique handles transaction and system failures.

- First, consider transaction failure. If the transaction fails at any time before db-pointer is updated, the old contents of the database are not affected.
- We can abort the transaction by just deleting the new copy of the database.
- Once the transaction has been committed, all the updates that it performed are in the database pointed to by dbpointer.
- Thus, either all updates of the transaction are reflected, or none of the effects are reflected, regardless of transaction failure.

Now consider the issue of system failure.

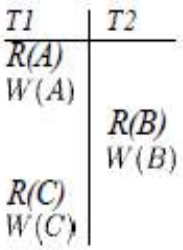
- Suppose that the system fails at any time before the updated db-pointer is written to disk. Then, when the system restarts, it will read db- pointer and will thus see the original contents of the database, and none of the effects of the transaction will be visible on the database.
- Next, suppose that the system fails after db- pointer has been updated on disk. Before the pointer is updated, all updated pages of the new copy of the database were written to disk.
- Again, we assume that, once a file is written to disk, its contents will not be damaged even if there is a system failure. Therefore, when the system restarts, it will read db-pointer and will thus see the contents of the database after all the updates performed by the transaction.

TRANSACTIONS AND SCHEDULES

A transaction is seen by the DBMS a'l a series, or *list,* of **actions.** The actions that can be executed by a transaction include **reads** and **writes** of *databaseobjects.* To keep our notation simple, we assume that an object O is always read into a program variable that is also named O. can therefore denote the action of a transaction *T* reading an object O as *RT(O);* similarly, we can denote writing as *HTT(O).* When the transaction *T* is clear from the context, we omit the subscript. In addition to reading and writing, each transaction *must* specify as its final action either **commit** (i.e., complete successfully) or **abort** (i.e., terminate and undo all the actions carried out thus far). *Abort T* denotes the action of aborting, and *Commit T* denotes *T* committing.

A **schedule** is a list of actions (reading, writing, aborting, or committing) from a set of transactions, and the order in which two actions of a transaction *T* appear in a schedule must be the same as the order in which they appear in *T.* Intuitively, a schedule represents an actual

or potential execution sequence. For example, the schedule in Figure shows an execution order for actions of two transactions *T1* and *T2*.



4. Concurrent Executions

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. However, there are two good reasons for allowing concurrency:

• ***Improved throughput and resource utilization:***

- A transaction consists of many steps. Some involve I/O activity; others involve CPU activity. The CPU and the disks in a computer system can operate in parallel. Therefore, I/O activity can be done in parallel with processing at the CPU.
- The parallelism of the CPU and the I/O system can therefore be exploited to run multiple transactions in parallel. While a read or write on behalf of one transaction is in progress on one disk, another transaction can be running in the CPU, while another disk may be executing a read or write on behalf of a third transaction.
- All of this increases the throughput of the system—that is, the number of transactions executed in a given amount of time. Correspondingly, the processor and disk utilization also increase; in other words, the processor and disk spend less time idle, or not performing any useful work.

• ***Reduced waiting time:***

- There may be a mix of transactions running on a system, some short and some long. If transactions run serially, a short transaction may have to wait for a preceding long transaction to complete, which can lead to unpredictable delays in running a transaction.
- If the transactions are operating on different parts of the database, it is

better to let them run concurrently, sharing the CPU cycles and disk accesses among them.

- Concurrent execution reduces the unpredictable delays in running transactions. Moreover, it also reduces the average response time: the average time for a transaction to be completed after it has been submitted.

Let T1 and T2 be two transactions that transfer funds from one account to another. Transaction T1 transfers $50 from account A to account B. It is defined as:

T_1: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

Transaction T2 transfers 10 percent of the balance from account A to account B. It is defined as:

T_2: read(A);
$temp := A * 0.1$;
$A := A - temp$;
write(A);
read(B);
$B := B + temp$;
write(B).

Suppose the current values of accounts A and B are $1000 and $2000, respectively. Suppose also that the two transactions are executed one at a time in the order T1followed by T2.This execution sequence appears in Figure below. In the figure, the sequence of instruction steps is in chronological order from top to bottom, with instructions of T1 appearing in the left column and instructions of T2 appearing in the right column. The final values of accounts A and B, after the execution in Figure below, takes place, are $855 and $2145, respectively. Thus, the total amount of money in accounts A and B—that is, the sum A + B—is preserved after the execution of both transactions

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

Schedule 1—a serial schedule in which T_1 is followed by T_2.

Similarly, if the transactions are executed one at a time in the order T2 followed by T1, then the corresponding execution sequence is that of Figure below. Again, as expected, the sum A+ B is preserved, and the final values of accounts A and B are $850 and $2150, respectively.

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

Schedule 2—a serial schedule in which T_2 is followed by T_1.

When the database system executes several transactions concurrently, the corresponding schedule no longer needs to be serial. If two transactions are running concurrently, the operating system may execute one transaction for a little while, then perform a context switch, execute the second transaction for some time, and then switch back to the first transaction for some time, and so on. With multiple transactions, the CPU time is shared among all the transactions.

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

Schedule 3—a concurrent schedule equivalent to schedule 1.

Not all concurrent executions result in a correct state. To illustrate, consider the schedule of Figure below:

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

Schedule 4—a concurrent schedule.

After the execution of this schedule, we arrive at a state where the final values of accounts A and B are $950 and $2100, respectively. This final state is an inconsistent state, since we have gained $50 in the process of the concurrent execution. Indeed, the sum A + B is not preserved by the execution of the two transactions.

If control of concurrent execution is left entirely to the operating system, many possible schedules, including ones that leave the database in an inconsistent state, such as the one just described, are possible. It is the job of the database system to ensure that any schedule that gets executed will leave the database in a consistent state. The concurrency-control component of the database system carries out this task.

5. Serializability:

A **serializable schedule** over a set *S* of cormnitted transactions is a schedulewhose effect on any consistent database instance is guaranteed to be identicalto that of some complete serial schedule over *S.* That is, the database instancethat results from executing the given schedule is identical to the database instancethat results from executing the transactions in *some* serial order.

As an example, the schedule shown in Figure is serializable. Even thoughthe actions of *T1* and *T2* are interleaved, the result of this schedule is equivalent .to running *T1* (in its entirety) and then running *T2.* Intuitively, *T1* 's read and write of *B* is not influenced by *T2's* actions on *A,* and the net effect is the same

if these actions are 'swapped' to obtain the serial schedule *Tl; T2.*

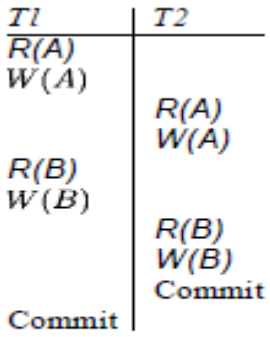


Figure 16.2 A Serializable Schedule

Two forms of serializabilty are

- Conflict Serializabilty
- View Serializability.

Conflict Serializability:

Let us consider a schedule S in which there are two consecutive instructions Ii and Ij ,of transactions Ti and Tj, respectively (i ≠ j). If Ii and Ij refer to different data items, then we can swap Ii and Ij without affecting the results of any instruction in the schedule. However, if Ii and Ij refer to the same data item Q, then the order of the two steps may matter. Since we are

dealing with only read and write instructions, there are four cases that we need to consider

1. Ii = read(Q), Ij = read(Q). The order of Ii and Ij does not matter, since the same value of Q is read by Ti and Tj, regardless of the order.
2. Ii= read(Q), Ij= write(Q).If Ii comes before Ij, then Ti does not read the value of Q that is written by Tj in instruction Ij. If Ij comes before Ii, then Ti reads the value of Q that is written by Tj. Thus , the order of Ii and Ij matters.
3. Ii= write(Q), Ij= read(Q). the order of Ii and Ij matters for reasons similar to those of the previous case.
4. Ii= write(Q), Ij= write(Q). since both instructions are write operations, the order of these instructions does not affect either Ti or Tj. However, the value obtained by the next read(Q) instruction of S is affected, since the result of only the latter of the two write instructions is preserved in the database. If there is no other write(Q) instruction after Ii and Ij in S, then the order of Ii and Ij directly affects the final value of Q in the database state that results from schedule S.

Thus, only in the case where both Ii and Ij are read instructions does the relative order of their execution not matter.

We say that Ii and Ij conflict if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation. To illustrate the concept of conflicting instructions, we consider schedule 3, in Fig above. The write(A) instruction of T1 conflicts with the read(A) instruction of T2. However, the write(A) instruction of T2 does not conflict with the read(B) instruction of T1, because the two instructions access different data items.

Let Ii and Ij be consecutive instructions of a schedule S. If Ii and Ij are instructions of different transactions and Ii and Ij do not conflict, then we can swap the order of Ii and Ij to produce a new schedule S'. We expect S to be equivalent to S'. Since all instructions appear in the same order in both schedules except for Ii and Ij, whose order does not matter.

Since the write (A) instruction of Tin schedule 3 does not conflict with the read (B) instruction of T1, we can swap these instructions to generate an equivalent schedule, schedule 5,shown in Figure below.

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

Schedule 5—schedule 3 after swapping of a pair of instructions.

Regardless of the initial system state, schedules 3 and 5 both produce the same final system state. We continue to swap nonconflicting instructions:

- Swap the read(B) instruction of T_1 with the read(A) instruction of T2

.

- Swap the write(B) instruction of T_1 with the write(A) instruction of T2
- Swap the write(B) instruction of T_1 with the read(A) instruction of T2.

The final result of these swaps, schedule 6 of Figure below, is a serial schedule

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

Schedule 6—a serial schedule that is equivalent to schedule 3.

This equivalence implies that, regardless of the initial system state, schedule 3 will produce the same final state as will some serial schedule.If a schedule S can be transformed into a schedule S' by a series of swaps of non conflicting instructions, we say that S and S' are conflict equivalent.

The concept of conflict equivalence leads to the concept of conflict serializability. We say that a schedule S is conflict serializable if it is conflict equivalent to a serial schedule.

Finally, consider schedule 7 of Figure below; it consists of only the significant operations (that is, the read and write)of transactions T3 and T4. This schedule is not conflict serializable, since it is not equivalent to either the serial schedule <T3,T4> or the serial schedule <T4,T3>. It is possible to have two schedules that produce the same outcome, but that are not conflict equivalent.

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

View Serializability:

Consider two schedules S and S' , where the same set of transactions participates in both schedules. The schedules S and S 'are said to be view equivalent if three conditions are met:

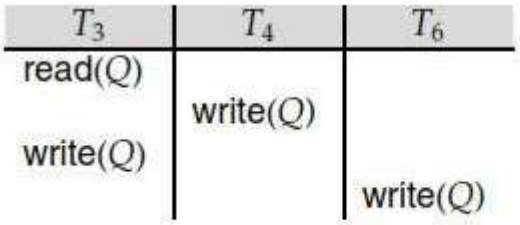
1. For each data item Q, if transaction Ti reads the initial value of Q in schedule S, then transaction Ti must, in schedule S , also read the initial value of Q.
2. For each data item Q, if transaction Ti executes read(Q) in schedule S, and if that value was produced by a write(Q) operation executed by transaction Tj, then the read(Q) operation of transaction Ti must, in schedule S value of Q that was produced by the same write(Q) operation of transaction Tj.
3. For each data item Q, the transaction (if any) that performs the final write(Q) operation in schedule S must perform the final write(Q)operation in schedule S'.

Conditions 1 and 2 ensure that each transaction reads the same values in both schedules and, therefore, performs the same computation. Condition 3, coupled with conditions 1 and 2, ensures that both schedules result in the same final system state.

In our previous examples, schedule 1 is not view equivalent to schedule 2, since, in schedule 1, the value of account A read by transaction T2 was produced by T1, whereas this case does not hold in schedule 2. However, schedule 1 is view

equivalent to schedule 3, because the values of account A and B read by transaction T2 were produced by T1 in both schedules.

The concept of view equivalence leads to the concept of view serializability. We say that a schedule S is view serializable if it is view equivalent to a serial schedule. As an illustration, suppose that we augment schedule 7 with transaction T6,and obtain schedule 9 in Figure below:



2 Schedule 9—a view-serializable schedule.

Schedule 9 is view serializable. Indeed, it is view equivalent to the serial schedule

<T3,T4,T6>, since the one read(Q) instruction reads the initial value of Q in both schedules, and T6 performs the final write of Q in both schedules.

Every conflict-serializable schedule is also view serializable, but there are viewserializable schedules that are not conflict serializable. Indeed, schedule 9 is not conflict serializable, since every pair of consecutive instructions conflicts, and, thus, no swapping of instructions is possible.

Observe that, in schedule 9, transactions T4 and T6 perform write(Q)operation without having performed a read(Q) operation. Writes of this sort are called blind writes. Blind writes appear in any view-serializable schedule that is not conflict serializable.

6. Recoverability

Now address the effect of transaction failures during concurrent execution. If Transaction Ti fails, for whatever reason, we need to undo the effect of this transaction to ensure the atomicity property of the transaction. In a system that allows concurrent execution, it is necessary also to ensure that any transaction Tj that is dependent on Ti (that is, Tj has read data written by Ti) is also aborted. To achieve this surety, we need to place restrictions on the type of schedules permitted in the system.

Recoverable Schedules:

Consider schedule 11 in Figure below:

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

Schedule 11.

in which T9 is a transaction that performs only one instruction: read(A). Suppose that the system allows T9 to commit immediately after executing the read(A) instruction. Thus, T9commits before T8 does. Now suppose that T8 fails before it commits. Since T9 has read the value of data item A written by T8 ,we must abort T9 to ensure transaction atomicity. However, T9 has already committed and cannot be aborted. Thus, we have a situation where it is impossible to recover correctly from the failure of T8.

Schedule 11, with the commit happening immediately after the read(A) instruction, is an example of a nonrecoverable schedule, which should not be allowed. Most database system require that all schedules be recoverable. A recoverable schedule is one where, for each pair of transactions Ti and T such that T reads a data item previously written by Ti , the commit operation of appears before the commit operation of Tj.

Cascadeless Schedules

Even if a schedule is recoverable, to recover correctly from the failure of a transaction Ti, we may have to roll back several transactions. Such situations occur if transactions have read data written by Ti. As an illustration, consider the partial schedule of figbelow:

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

Schedule 12.

Transaction T10 writes a value of A that is read by transaction T11. Transaction T11 writes a values of A that is tead by instrction T12, suppose tht at this point, T10 fails. T10 must be rolled back. Since T11 is dependent on T10, T11 must be

rolled back. Since T12 is dependent on T11, T12 must be rolled back. This phenonmenon, in which a single transaction failure leads to a series of transaction Rollbacks, is called Cascading Rollback.

Cascading rollback is undesirable, since it leads to the undoing of a significant amount of work. It is desirable to restrict the schedules to those where cascading rollbacks cannot occur. Such schedules are called cascadeless schedules. Formally, a cascadeless schedule is one where, for each pair of transactions Ti such that Tj reads a data item previously written by Ti , the commit operation of Ti appears before the read operation of T j. It is easy to verify that every cascadeless schedule is also recoverable.

7. Implementation of Isolation

There are various concurrency-control schemes that we can use to ensure that, even when multiple transactions are executed concurrently, only acceptable schedules are generated, regardless of how the operating-system time-shares resources (such as CPU time) among the transactions. As a trivial example of a concurrency-control scheme, consider this scheme:

A transaction acquires a lock on the entire database before it starts and releases the lock after it has committed. While a transaction holds a lock, no other transaction is allowed to acquire the lock, and all must therefore wait for the lock to be released. As a result of the locking policy, only one transaction can execute at a time. Therefore, only serial schedules are generated. These are trivially serializable, and it is easy to verify that they are cascadeless as well.

A concurrency-control scheme such as this one leads to poor performance, since it forces transactions to wait for preceding transactions to finish before they can start. In other words, it provides a poor degree of concurrency. concurrent execution has several performance benefits. The goal of concurrency-control schemes is to provide a high degree of concurrency, while ensuring that all schedules that can be generated are conflict or view serializable, and are cascadeless.

8.Testing for Serializability

We now present a simple and efficient method for determining conflict serializability of a schedule. Consider a schedule S. We construct a directed graph, called a precedence graph,fromS. 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 Ti →Tj for which one of three conditions holds:

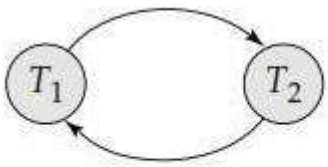
1. Ti executes write(Q) before Tj executes read(Q).
2. Ti executes read(Q) before Tj executes write(Q).
3. Ti executes write(Q) before Tj executes write(Q).

If an edge Ti → Tj exists in the precedence graph, then, in any serial schedule S' equivalent to S, must appear before Tj.

Ex: The precedence graph for schedule 1 in Figure(a) below a contains the single edge T1→ T2, since all the instructions of T1 are executed before the first instruction of T2 is executed. Similarly, Figure(b) shows the precedence graph for schedule 2 with the single edge T2→ T1, since all the instructions of T2 are executed before the first instruction of T1 is executed.

Precedence graph for (a) schedule 1 and (b) schedule 2.

The precedence graph for schedule 4 appears in Figure below:

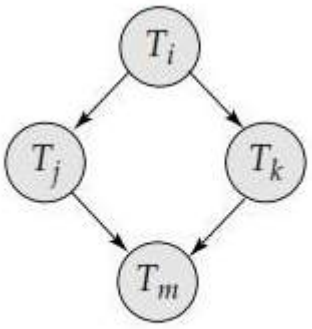


Precedence graph for schedule 4.

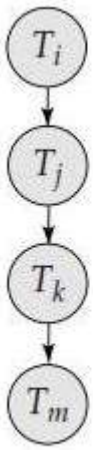
It contains the edge T1 →T2, because T1 executes read(A)before T2 executes write(A). It also contains the edge T2→ T1,becauseT2 executes read(B) before T1executes write(B). If the precedence graph for S has a cycle, then schedule S is not conflict serializable.If the graph contains no cycles, then the schedule S is conflict serializable.

A serializability order of the transactions can be obtained through topological sorting, which determines a linear order consistent with the partial order of the precedence graph. There are, in general, several possible linear orders

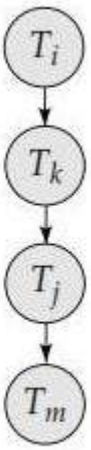
that can be obtained through a topological sorting. For example, the graph of Figure(a) has the two acceptable linear orderings shown in Figures(b) and (c).



(a)



(b)



(c)

Thus, to test for conflict serializability, we need to construct the precedence graph and to invoke a cycle-detection algorithm. Cycle-detection algorithms, such as those based on depth-first search, require on the order of n^2operations, where n is the number of vertices in the graph (that is, the number of transactions). Thus, we have a practical scheme for determining conflict serializability.

Testing for view serializability is rather complicated. In fact, it has been shown that the problem of testing for view serializability is itself NP-complete. Thus, almost certainly there exists no efficient algorithm to test for view serializability.

11.Lock-Based Protocols:

One way to ensure serializability 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.

Locks

There are various modes in which a data item may be locked. In this section, we restrict our attention to two modes:

1. **Shared:** If a transaction Ti has obtained a shared-mode lock (denoted by S) on item Q, then Ti can read, but cannot write, Q.

2. **Exclusive**: If a transaction Ti has obtained an exclusive-mode lock (denoted by X) on item Q, then Ti can both read and write Q.

To access a data item Q, transaction Ti must first lock that item in an appropriate mode, 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.

Let transaction Ti is having Shared-lock on data item A and transaction Tj is trying to apply Exclusive lock on A, Then concurrency-control manager verifies lock compatibility and applies the lock if it is compatible.

	S	X
S	true	false
X	false	false

Lock-compatibility matrix comp.

As per the lock compatibility matrix on a data item If shared lock is applied then other transactions are allowed to apply only shared lock.

On the data item if exclusive lock was applied then no other transaction is allowed apply shared or exclusive lock on the data item.

A transaction requests a shared lock on data item Q by executing the lock-S(Q) instruction. Similarly, a transaction requests an exclusive lock through the lock-X(Q) instruction. A transaction can unlock a data item Q by the unlock(Q) instruction.

Transaction Ti 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, for a transaction to unlock a data item immediately after its final access of that data item is not always desirable, since serializability may not be ensured.

Ex: Let A and B be two accounts that are accessed by transactions T1 and T2.Transaction T1 transfers $50 from account B to account A. Transaction T2 displays the total amount of money in accounts A and B—that is, the sum A + B.

Suppose that the values of accounts A and B are $100 and $200, respectively. If these two transactions are executed serially, either in the order T1, T2 or the order T2,T1, then transaction T2 will display the value $300. If, however, these transactions are executed concurrently, then schedule 1, in Figure below is possible. In this case, transaction T2 displays $250, which is incorrect. The reason for this mistake is that the transaction T1 unlocked data item B too early, as a result of which T2 saw an inconsistent state.

Suppose now that unlocking is delayed to the end of the transaction. Transaction T3 corresponds to T1 with unlocking delayed. Transaction T4 corresponds to T2 with unlocking delayed (Figure below).

```
T3: lock-X(B);
    read(B);
    B := B − 50;
    write(B);
    lock-X(A);
    read(A);
    A := A + 50;
    write(A);
    unlock(B);
    unlock(A).
```

```
T4: lock-S(A);
    read(A);
    lock-S(B);
    read(B);
    display(A + B);
    unlock(A);
    unlock(B).
```

Locking can lead to an undesirable situation. Consider the partial schedule of Figure below for T3 and T4:

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

SinceT3 is holding an exclusive-mode lock on B and T4 is requesting a shared-mode lock on B, T3 is waiting for T4 to unlock. Similarly, since T4 is holding a shared-mode lock on A and T3 is requesting an exclusive-mode lock on A, T3 is waiting for T4 to unlock A. Thus, we have arrived at state where neither of these transactions can ever

proceed with its normal execution. This situation is called deadlock. When deadlock occurs, the system must roll back one of the two transactions. Once a transaction has been rolled back, the data items that were locked by that transaction are unlocked. These data items are then available to the other transaction, which can continue with its execution.

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.

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 follow 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 → relation is acyclic.

Granting of Locks

We can avoid starvation of transactions by granting locks in the following manner: When a transaction Ti 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.

Thus, a lock request will never get blocked by a lock request that is made later.

The Two-Phase Locking Protocol

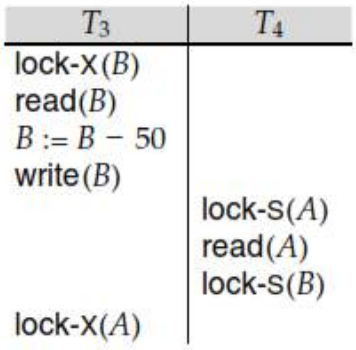
One protocol that ensures serializability is the two-phase locking protocol. This protocol requires that each transaction issue lock and unlock requests in two phases:

1. **Growing 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.

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. For example, transactions T3 and T4 are two phase. On the other hand, transactions T1 and T2 are not two phase. Note that the unlock instructions

do not need to appear at the end of the transaction. For example, in the case of transaction T3,we could move the unlock(B) instruction to just after the lock-X(A) instruction, and still retain the two- phase locking property.

Two-phase locking does not ensure freedom from deadlock. Observe that transactions

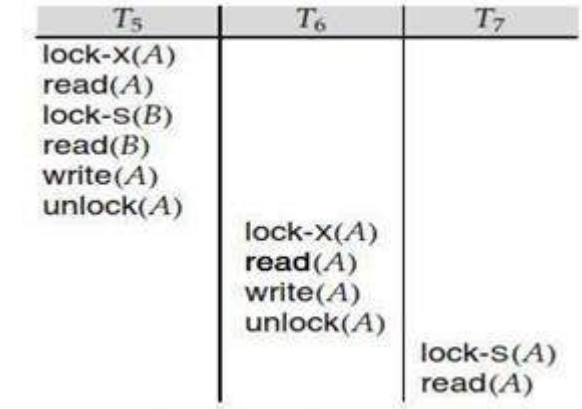


T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

T3 and T4 are two phase, but they are deadlocked.

T3 and T4 are two phase, but they are deadlocked.

In addition to being serializable, schedules should be cascadeless. Cascading rollback may occur under two-phase locking. As an illustration, consider the partial schedule of Figure below



T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

Partial schedule under two-phase locking.

Each transaction observes the two-phase locking protocol, but the failure of T5 after the read(A) step of T7 leads to cascading rollback of T6 and T7.

In order to avoid the problems of two-phase locking protocol, there exist variants of it, they are

Strict Two-phase Locking: 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 phase, 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: Another variant of two-phase locking is the rigorous two-phase locking protocol, which requires that all locks be held until the transaction commits. We can easily verify that, with rigorous two-phase locking, transactions can be serialized in the order in which they commit. Most database systems implement either strict or rigorous two- phase locking.

Consider the following two transactions, for which we have shown only some of the significant read and write operations:

T_8: read(a_1);
read(a_2);
...
read(a_n);
write(a_1).

T_9: read(a_1);
read(a_2);
display($a_1 + a_2$).

If we employ the two-phase locking protocol, then T8 must lock a1 in exclusive mode. Therefore, any concurrent execution of both transactions amounts to a serial execution.

Notice, however, that T8 needs an exclusive lock on a only at the end of its execution, when it writes a1.Thus, if T8 could initially lock a1 in shared mode, and then could later change the lock to exclusive mode, we could get more concurrency, since T8 and T9 could access a1 and a2 simultaneously.

Strict two-phase locking and rigorous two-phase locking (with lock conversions) are used extensively in commercial database systems. A simple but widely used scheme automatically generates the appropriate lock and unlock instructions for a transaction, on the basis of read and write requests from the transaction:

- When a transaction Ti issues a read(Q) operation, the system issues a lock-S(Q) instruction followed by the read(Q) instruction.
- When Ti issues a write(Q) operation, the system checks to see whether Ti already holds a shared lock on Q. If it does, then the system issues

an upgrade(Q) instruction, followed by the write(Q) instruction. Otherwise, the system issues a lock- X(Q) instruction, followed by the write(Q) instruction.

- All locks obtained by a transaction are unlocked after that transaction commits or aborts.

12.Timestamp-Based Protocols

Another method for determining the serializability order is to select an ordering among transactions in advance. The most common method for doing so is to use a timestamp- ordering scheme.

Timestamps

With each transaction Ti in the system, we associate a unique fixed timestamp, denoted by TS(Ti). This timestamp is assigned by the database system before the transaction Ti starts execution. If a transaction Ti has been assigned timestamp TS(Ti), and a new transaction Tj enters the system, then TS(Ti) < TS(Tj). 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(Ti)

<TS(Tj), then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction Ti appears before transaction Tj. 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.

These timestamps are updated whenever a new read(Q)orwrite(Q) instruction is executed.

The Timestamp-Ordering Protocol

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 Ti issues read(Q).

a. 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.

b. If TS(Ti) ≥ W-timestamp(Q), then the read operation is executed, and Rtimestamp(Q) is set to the maximum of R-timestamp(Q)andTS(T).

2. Suppose that transaction Ti issues write(Q).

a. 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 system rejects the write operation and rolls Ti back.

b. If TS(Ti) < W-timestamp(Q), then Ti is attempting to write an obsolete value of Q. Hence, the system rejects this write operation and rolls Ti back.

c. Otherwise, the system executes the write operation and sets W-time-stamp(Q)toTS(Ti).

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

3. To illustrate this protocol, we consider transactions T14 and T15 displays the contents of accounts A and B:

T_{14}: read(B);
read(A);
display($A + B$).

Transaction T15 transfers $50 from account A to account B, and then displays the contents of both

T_{15}: read(B);
$B := B - 50$;
write(B);
read(A);
$A := A + 50$;
write(A);
display($A + B$).

In presenting schedules under the timestamp protocol, we shall assume that a transaction is assigned a timestamp immediately before its first instruction. Thus, in schedule3 of Figure below, TS(T14) < TS(T15), and the schedule is possible under the timestamp protocol.

T_{14}	T_{15}
read (B)	
	read (B)
	$B := B - 50$
	write (B)
read (A)	
	read (A)
display $(A + B)$	
	$A := A + 50$
	write (A)
	display $(A + B)$

The protocol can generate schedules that are not recoverable. However, it can be extended to make the schedules recoverable, in one of several ways:

Recoverability and cascadelessness can be ensured by performing all writes together at the end of the transaction. The writes must be atomic in the following sense: While the writes are in progress, no transaction is permitted to access any of the data items that have been written.

Recoverability and cascadelessness can also be guaranteed by using a limited form of locking, whereby reads of uncommitted items are postponed until the transaction that updated the item commits.

Recoverability alone can be ensured by tracking uncommitted writes, and allowing a transaction Ti to commit only after the commit of any transaction that wrote a value that Ti read. Commit dependencies can be used for this purpose.

Thomas' Write Rule

We now present a modification to the timestamp-ordering protocol that allows greater potential concurrency than does the protocol. Let us consider schedule of Figure below and apply the timestamp-ordering protocol.

T_{16}	T_{17}
read (Q)	
	write (Q)
write (Q)	

Since T16 starts before T17, we shall assume that TS(T16) < TS(T17). The read(Q) operation of T16 succeeds, as does the write(Q) operation of T16 attempts its write(Q)operation, we find that TS(T16)) < W-timestamp(Q), since W-timestamp(Q)=TS(T17).Thus, the write(Q) by T16 is rejected and transaction T16 must be rolled back.

Although the rollback of T16 is required by the timestamp-ordering protocol, it is unnecessary. Since T 17 has already written Q, the value that T16 is attempting to write is one that will never need to be read. Any transaction Ti with TS(Ti)< TS(T17) that attempts a read(Q) will be rolled back, since TS(Ti)) < W-timestamp(Q). Any transaction Tj with TS(Tj)

>TS(T17)must read the value of Q written by T17,ratherthan the value written by T16

This observation leads to a modified version of the timestamp-ordering protocol in which obsolete write operations can be ignored under certain circumstances. The protocol rules for read operations remain unchanged. The protocol rules for write operations, however, are slightly different from the timestamp-ordering protocol. The modification to the timestamp- ordering protocol, called Thomas' write rule, is this: Suppose that transaction Ti issues write(Q).

1. If TS(Ti) < R-timestamp(Q), then the value of Q that Ti is producing was previously needed, and it had been assumed that the value would never be produced. Hence, the system rejects the write operation and rolls Ti back.

2. If TS(Ti) < W-timestamp(Q), then Ti is attempting to write an obsolete value of Q. Hence, this write operation can be ignored.

3. Otherwise, the system executes the write operation and sets W-timestamp(Q) to

TS(Ti).

The difference between time-stamp protocol and Thomas write protocol lies in the second rule. The time stamp ordering protocol requires Ti is rolled back if Ti issues write(Q) and TS(Ti) < W-timestamo(Q). however, herem in those case where TS(Ti)≥ R- timestmp(Q). we ignore the obsolete write.

13.Validation-Based Protocols

Validation phase is also known as optimistic concurrency control technique. In the validation based protocol, the transaction is executed in the following three phases:

1. **Read phase:** In this phase, the transaction T is read and executed. It is used to read the value of various data items and stores them in temporary local variables. It

can perform all the write operations on temporary variables without an update to the actual database.

2. **Validation phase:** In this phase, the temporary variable value will be validated against the actual data to see if it violates the serializability.
3. **Write phase:** If the validation of the transaction is validated, then the temporary results are written to the database or system otherwise the transaction is rolled back.

Here each phase has the following different timestamps:

Start(Ti): It contains the time when Ti started its execution.

Validation (T_i): It contains the time when Ti finishes its read phase and starts its validation phase.

Finish(Ti): It contains the time when Ti finishes its write phase.

- This protocol is used to determine the time stamp for the transaction for serialization using the time stamp of the validation phase, as it is the actual phase which determines if the transaction will commit or rollback.
- Hence TS(T) = validation(T).
- The serializability is determined during the validation process. It can't be decided in advance.
- While executing the transaction, it ensures a greater degree of concurrency and also less number of conflicts.
- Thus it contains transactions which have less number of rollbacks.

As an illustration, consider transactions T14 and T15 shown below:

T_{14}	T_{15}
read(B)	
	read(B)
	$B := B - 50$
	read(A)
	$A := A + 50$
read(A)	
⟨*validate*⟩	
display($A + B$)	
	⟨*validate*⟩
	write(B)
	write(A)

Suppose that TS(T14)< TS(T15). Then, the validation phase succeeds in the above schedule shown in Figure. Note that the writes to the actual variables are performed

only after the validation phase of T15.Thus, T14 reads the old values of B and A, and this schedule is serializable.

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.

14.Multiple Granularity

Multiple Granularity:

- It can be defined as hierarchically breaking up the database into blocks which can be locked.
- The Multiple Granularity protocol enhances concurrency and reduces lock overhead.
- It maintains the track of what to lock and how to lock.
- It makes easy to decide either to lock a data item or to unlock a data item. This type of hierarchy can be graphically represented as a tree.

For example: Consider a tree which has four levels of nodes.

- The first level or higher level shows the entire database.
- The second level represents a node of type area. The higher level database consists of exactly these areas.
- The area consists of children nodes which are known as files. No file can be present in more than one area.
- Finally, each file contains child nodes known as records. The file has exactly those records that are its child nodes. No records represent in more than one file.
- Hence, the levels of the tree starting from the top level are as follows:
    1. Database
    2. Area
    3. File
    4. Record

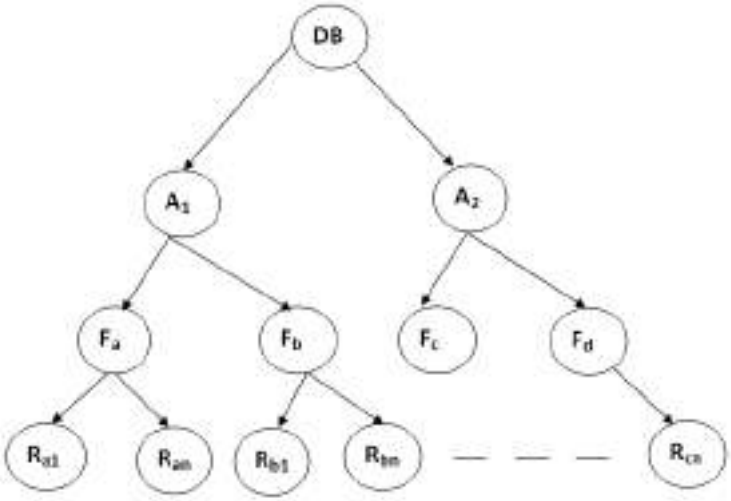


Figure: Multi Granularity tree Hierarchy

In this example, the highest level shows the entire database. The levels below are file, record, and fields.
There are three additional lock modes with multiple granularity:

Intention Mode Lock

Intention-shared (IS): It contains explicit locking at a lower level of the tree but only with shared locks.

Intention-Exclusive (IX): It contains explicit locking at a lower level with exclusive or shared locks.

Shared & Intention-Exclusive (SIX): In this lock, the node is locked in shared mode, and some node is locked in exclusive mode by the same transaction.

Compatibility Matrix with Intention Lock Modes: The below table describes the compatibility matrix for these lock modes:

	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	X
IX	✓	✓	X	X	X
S	✓	X	✓	X	X
SIX	✓	X	X	X	X
X	X	X	X	X	X

It uses the intention lock modes to ensure serializability. It requires that if a transaction attempts to lock a node, then that node must follow these protocols:

- Transaction T1 should follow the lock-compatibility matrix.
- Transaction T1 firstly locks the root of the tree. It can lock it in any mode.

- If T1 currently has the parent of the node locked in either IX or IS mode, then the transaction T1 will lock a node in S or IS mode only.
- If T1 currently has the parent of the node locked in either IX or SIX modes, then the transaction T1 will lock a node in X, SIX, or IX mode only.
- If T1 has not previously unlocked any node only, then the Transaction T1 can lock a node.
- If T1 currently has none of the children of the node-locked only, then Transaction T1 will unlock a node.

Observe that in multiple-granularity, the locks are acquired in top-down order, and locks must be released in bottom-up order.

- If transaction T1 reads record R_{a9} in file F_a, then transaction T1 needs to lock the database, area A_1 and file F_a in IX mode. Finally, it needs to lock R_{a2} in S mode.
- If transaction T2 modifies record R_{a9} in file F_a, then it can do so after locking the database, area A_1 and file F_a in IX mode. Finally, it needs to lock the R_{a9} in X mode.
- If transaction T3 reads all the records in file F_a, then transaction T3 needs to lock the database, and area A in IS mode. At last, it needs to lock F_a in S mode.
- If transaction T4 reads the entire database, then T4 needs to lock the database in S mode.

15.Recovery and Atomicity

Consider again our simplified banking system and transaction T that transfers $50 from account A to account B, with initial values of A and B being $1000 and $2000, respectively. Suppose that a system crash has occurred during the execution of Ti, after output(B_A) has taken place, but before output(B_B) was executed, where B_A and B_B denote the buffer blocks on which A and B reside. Since the memory contents were lost, we do not know the fate of the transaction; thus, we could invoke one of two possible recovery procedures:

- Re-execute Ti: This procedure will result in the value of A becoming $900, rather than $950. Thus, the system enters an inconsistent state.
- Do not re-execute Ti: The current system state has values of $950 and $2000 for A and B, respectively. Thus, the system enters an inconsistent state.

In either case, the database is left in an inconsistent state, and thus this simple recovery scheme does not work. The reason for this difficulty is that we have modified the database without having assurance that the transaction will indeed commit. Our goal is to perform either all or no database modifications made by Ti. However, if Ti performed multiple database modifications, several output operations may be required, and a failure may occur after some of these modifications have been made, but before all of them are made.

To achieve our goal of atomicity, we must first output information describing the modifications to stable storage, without modifying the database itself.

16.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** is the unique identifier of the transaction that performed the write operation.
- **Data-item identifier** is the unique identifier of the data item written. Typically, it is the location on disk of the data item.
- **Old value** is the value of the data item prior to the write.
- **New value** 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.

We denote the various types of log records as:

- <Ti start>.Transaction Ti has started.
- <Ti, Xj, V1, V2>.Transaction Ti has performed a write on data item Xj before the write, and will have value V12after the write.
- <Ti commit>.Transaction Ti has committed.
- <Ti abort>.Transaction Ti has aborted.

Whenever a transaction performs a write, it is essential that the log record for that write be created before the database is modified. Once a log record exists, we can output the modification to the database if that is desirable. Also, we have the ability to undo a modification that has already been output to the database. We undo it by using the old-value field in log records. For log records to be useful for recovery from system and disk failures, the log must reside in stable storage. For now, we assume that every log record is written to the end of the log on stable storage as soon as it is created.

17.Recovery with Concurrent Transactions

Regardless of the number of concurrent transactions, the system has a single disk buffer and a single log. All transactions share the buffer blocks. We allow immediate modification, and permit a buffer block to have data items updated by one or more transactions.

Interaction with Concurrency Control

The recovery scheme depends greatly on the concurrency-control scheme that is used. To roll back a failed transaction, we must undo the updates performed by the transaction. Suppose that a transaction Ti has to be rolled back, and a data item Q that was updated by T0 has to be restored to its old value. Using the log-based schemes for recovery, we restore the value by using the undo information in a log record. Suppose now that a second transaction T1 has performed yet another update on Q before T0 is rolled back. Then, the update performed by T1 will be lost if T0 is rolled back.

Therefore, we require that, if a transaction T has updated a data item Q, no other transaction may update the same data item until T has committed or been rolled back. We can ensure this requirement easily by using strict two-phase locking—that is, two-phase locking with exclusive locks held until the end of the transaction.

Recovery Algorithm

The recovery algorithm used in in two scenarios; first one is transaction rollback and second one is recovery after system crash.

Transaction Rollback

First consider transaction rollback during normal operation (that is, not during recovery from a system crash). Rollback of a transaction Ti is performed as follows:

1. The log is scanned backward, and for each log record of Ti of the form <Ti , Xj , V1, V2> that is found:

    a. The value V1 is written to data item Xj, and

    b. Aspecial redo-only log record<Ti ,Xj , V1>is written to the log, where V1 is the value being restored to data item Xj during the rollback. These log records are sometimes called **compensation log records**. Such records do not need undo information, since we never need to undo such an undo operation. We shall explain later how they are used.

2. Once the log record <Ti start> is found the backward scan is stopped, and a log record
<Ti abort> is written to the log.

Observe that every update action performed by the transaction or on behalf of the transaction, including actions taken to restore data items to their old value, have now been recorded in the log.

Recovery After a System Crash

Recovery actions, when the database system is restarted after a crash, take place in two

phases:

1)REDO PHASE 2) UNDO PHASE

3. In the redo phase, the system replays updates of all transactions by scanning the log forward from the last checkpoint. The log records that are replayed include log records for transactions that were rolled back before system crash, and those that had not committed when the system crash occurred. This phase also determines all transactions that were incomplete at the time of the crash, and must therefore be rolled back. Such incomplete transactions would either have been active at the time of the checkpoint, and thus would appear in the transaction list in the checkpoint record, orwould have started later; further, such incomplete transactions would have neither a<Ti abort> nor a <Ti commit> record in the log.

The specific steps taken while scanning the log are as follows:

a. The list of transactions to be rolled back, undo-list, is initially set to the list L in the <checkpoint L> log record.
b. Whenever a normal log record of the form <Ti , Xj , V1, V2>, or a redo-only log record of the form <Ti , Xj , V2> is encountered, the operation is redone; that is, the value V2 is written to data item Xj .

c. Whenever a log record of the form <Ti start> is found, Ti is added to undo-list.
d. Whenever a log record of the form <Ti abort> or <Ti commit> is found, Ti isremoved from undo-list.

At the end of the redo phase, undo-list contains the list of all transactions that are incomplete, that is, they neither committed nor completed rollback before the crash.

4. In the undo phase, the system rolls back all transactions in the undo-list. It performs rollback by scanning the log backward from the end.

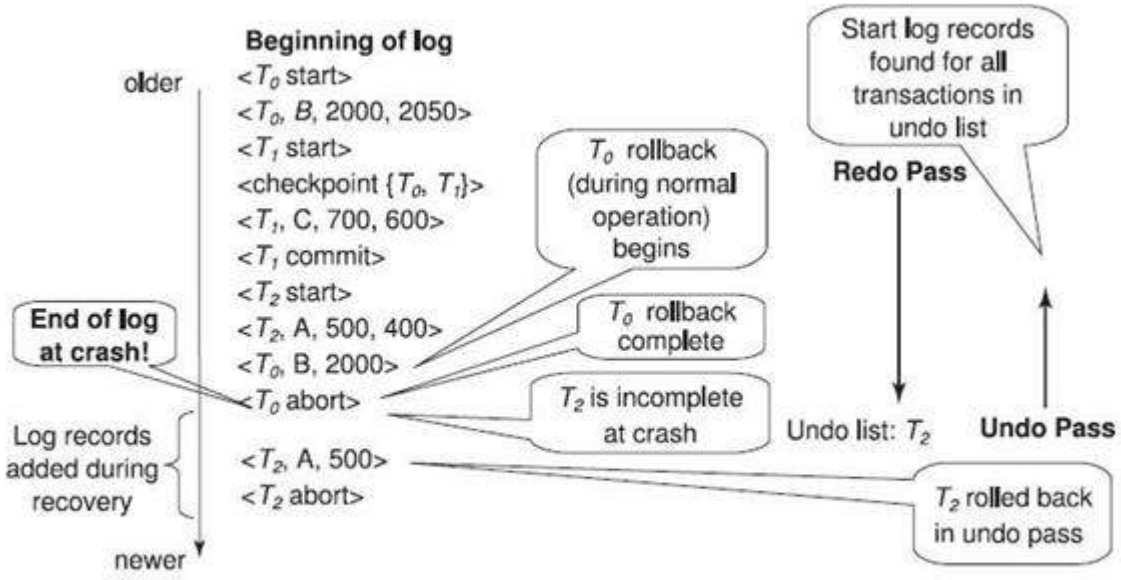
a. Whenever it finds a log record belonging to a transaction in the undo list, it performs undo actions just as if the log record had been found during the rollback of a failed transaction.

b. When the system finds a <Ti start> log record for a transaction Ti in undo-list, it writes a <Ti abort> log record to the log, and removes Ti from undo-list.

c. The undo phase terminates once undo-list becomes empty, that is, the system has found <$T_i$start>log records for all transactions that were initially in undo-list.

After the undo phase of recovery terminates, normal transaction processing can resume.

EXAMPLE



Example of logged actions, and actions during recovery.

The above figure shows an example of actions logged during normal operation, and actions performed during failure recovery. In the log shown in the figure, transaction T1 had committed, and transaction T0 had been completely rolled back, before the system crashed. Observe how the value of data item B is restored during the rollback of T0. Observe also the checkpoint record, with the list of active transactions containing T0 and T1.

When recovering from a crash, in the redo phase, the system performs a redo of all operations after the last checkpoint record. In this phase, the list undo-list initially contains T0 and T1; T1 is removed first when its commit log record is found, while T2 is added when its start log record is found. Transaction T0 is removed from undo-list when its abort log record is found, leaving only T2 in undo-list. The undo phase scans the log backwards from the end, and when it finds a log record of T2 updating A, the old value of Ais restored, and a redo-only log record written to the log. When the start record for T2 is found, an abort record is added for T2. Since undo-list contains no more transactions, the undo phase terminates, completing recovery.

TCL Commands – Save point Commit and Roll back:

COMMIT command:

`COMMIT` command is used to permanently save any transaction into the database.

When we use any DML command like `INSERT`, `UPDATE` or `DELETE`, the changes made by these commands are not permanent, until the current session is closed, the changes made by these commands can be rolled back.

To avoid that, we use the `COMMIT` command to mark the changes as permanent.

Following is commit command's syntax,

```
COMMIT;
```

ROLLBACK command :

This command restores the database to last commited state. It is also used with `SAVEPOINT` command to jump to a savepoint in an ongoing transaction.

If we have used the `UPDATE` command to make some changes into the database, and realise that those changes were not required, then we can use the `ROLLBACK` command to rollback those changes, if they were not commited using the `COMMIT` command.

Following is rollback command's syntax,

```
ROLLBACK TO savepoint_name;
```

SAVEPOINT command:

`SAVEPOINT` command is used to temporarily save a transaction so that you can rollback to that point whenever required.

Following is savepoint command's syntax,

SAVEPOINT savepoint_name;

In short, using this command we can name the different states of our data in any table and then rollback to that state using the ROLLBACK command whenever required

Using Savepoint and Rollback:

Following is the table **class**,

id	name

1	Abhi
2	Adam
4	Alex

Lets use some SQL queries on the above table and see the results.

```
INSERT INTO class VALUES(5, 'Rahul');

COMMIT;

UPDATE class SET name = 'Abhijit' WHERE id = '5';

SAVEPOINT A;

INSERT INTO class VALUES(6, 'Chris');

SAVEPOINT B;

INSERT INTO class VALUES(7, 'Bravo');

SAVEPOINT C

SELECT * FROM class;
```

The resultant table will look like,

Now let's use the ROLLBACK command to roll back the state of data to the **savepoint B**.

```
ROLLBACK TO B;

SELECT * FROM class;
```

Now our **class** table will look like,

id	**Name**

1	Abhi
2	Adam
4	Alex
5	Abhijit
6	Chris

Now let's again use the ROLLBACK command to roll back the state of data to the **savepoint A**

```
ROLLBACK TO A;

SELECT * FROM class;
```

Now the table will look like,

id	**name**
1	Abhi
2	Adam
4	Alex
5	Abhijit