mood-book

UNIT-4

1. Transaction Concept

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.

Ex: SQL, COBOL, C, C++, or Java

Where it is delimited by statements (or function calls) of the form begin transaction and end transaction. The transaction consists of all operations executed between the begin transaction and end transaction.

To ensure integrity of the data, we require that the database system maintain the following properties of the transactions:

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

Consistency: Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database.

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

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

These properties are often called the ACID properties; the acronym is derived from the first letter of each of the four properties.

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:

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

Let us now consider each of the ACID requirements. (For ease of presentation, we consider them in an order different from the order A-C-I-D).

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.

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.

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.

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.

2. Transaction State

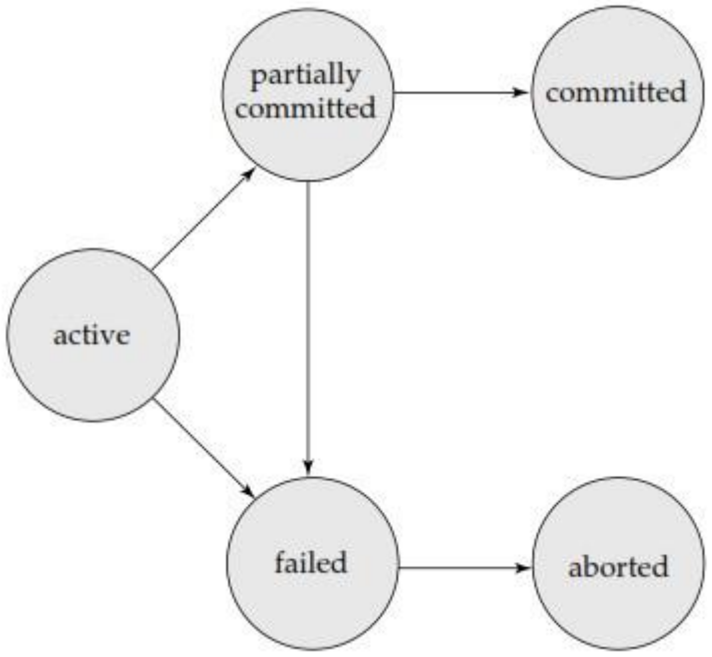
Aborted: A transaction may not always complete its execution successfully. Such a transaction is termed aborted. If we are to ensure the atomicity property, an aborted transaction must have no effect on the state of the database.

Rolledback: The aborted transaction made to the database must be undone. Once the changes caused by an aborted transaction have been undone, we say that the transaction has been rolled back. It is part of the responsibility of the recovery scheme to manage transaction aborts.

Committed: A transaction that completes its execution successfully is said to be committed. A committed transaction that has performed updates transforms the database into a new consistent state, which must persist even if there is a system failure.

Once a transaction has committed, we cannot undo its effects by aborting it. The only way to undo the effects of a committed transaction is to execute a compensating transaction. For instance, if a transaction added $20 to an account, the compensating transaction would subtract $20 from the account. However, it is not always possible to create such a compensating transaction. Therefore, the responsibility of writing and executing a compensating transaction is left to the user, and is not handled by the database system.

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 has committed only if it has entered the committed state. Similarly, we say that a transaction has aborted only if it has entered the aborted state. A transaction is said to have terminated if has either committed or aborted.

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.

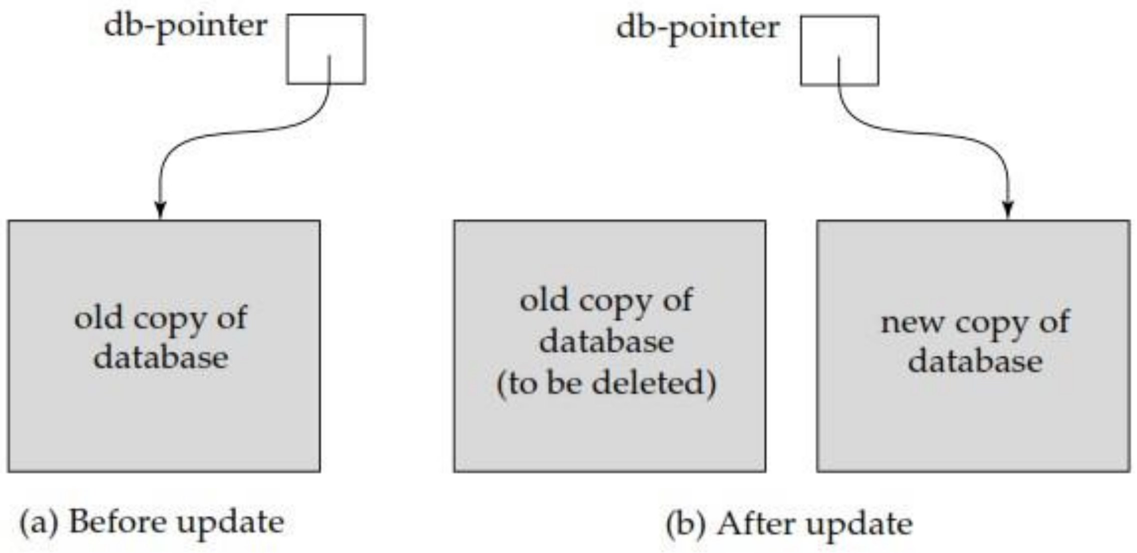
A transaction enters the failed state after the system determines that the transaction can no longer proceed with its normal execution (for example, because of hardware or logical errors). Such a transaction must be rolled back. Then, it enters the aborted state. At this point, the system has two options:

- 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.* This scheme, which 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.

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. 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 following Figure depicts the scheme, showing the database state before and after the update.



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.

The implementation actually depends on the write to db-pointer being atomic; that is, either all its bytes are written or none of its bytes are written. If some of the bytes of the pointer were updated by the write, but others were not, the pointer is meaningless, and neither old nor new versions of the database may be found when the system restarts. Luckily, disk systems provide atomic updates to entire blocks, or least to a disk sector. In other words, the disk system guarantees that it will update db-pointer atomically, as long as we make sure that db-pointer lies entirely in a single sector, which we can ensure by storing db-pointer at the beginning of a block. Thus, the atomicity and durability properties of transactions are ensured by the shadow-copy implementation of the recovery-management component.

Unfortunately, this implementation is extremely inefficient in the context of large databases, since executing a single transaction requires copying the entire database. Furthermore, the implementation does not allow transactions to execute concurrently with one another. There are practical ways of implementing atomicity and durability that are much less expensive and more powerful.

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.

The motivation for using concurrent execution in a database is essentially the same as the motivation for using multiprogramming in an operating system.

The database system must control the interaction among the concurrent transactions to prevent them from destroying the consistency of the database. It does so through a variety of mechanisms called concurrency-control schemes.

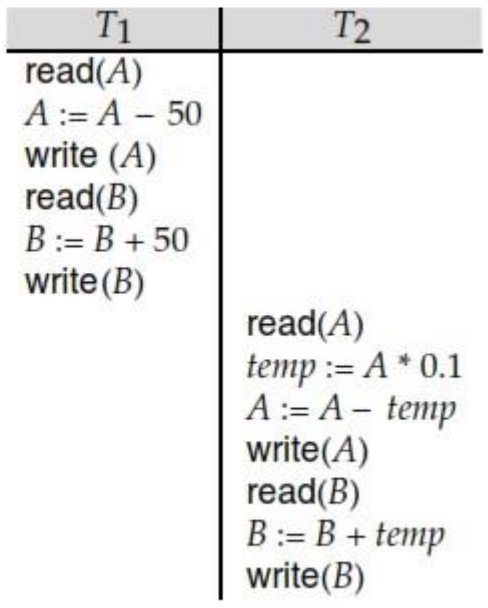
Consider again the simplified banking system, which has several accounts, and a set of transactions that access and update those accounts. Let T1 and T2 be two transactions that transfer funds from one account to another. Transaction T1 transfers $50 from account A to account B. It is defined as:

```
$T_1$: 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($B$).
```

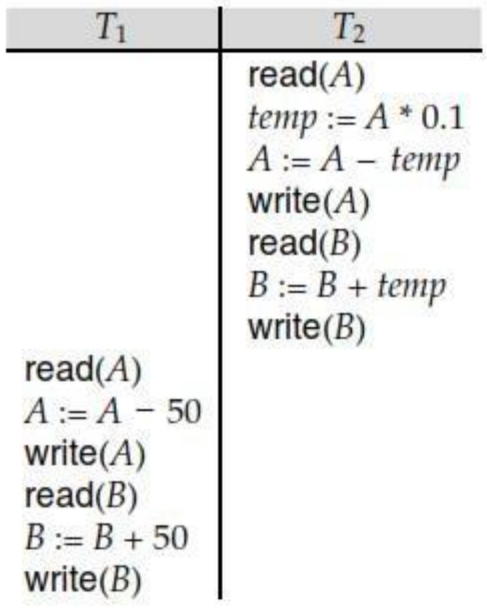
Suppose the current values of accounts A and B are $1000 and $2000, respectively. Suppose also that the two transactions are executed one at a time in the order T1followed by T2.This execution sequence appears in Figure below. In the figure, the sequence of instruction steps is in chronological order 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.

The execution sequences just described are called schedules. They represent the chronological order in which instructions are executed in the system. Clearly, a schedule for a set of transactions must consist of all instructions of those transactions, and must preserve the order in which the instructions appear in each individual transaction.

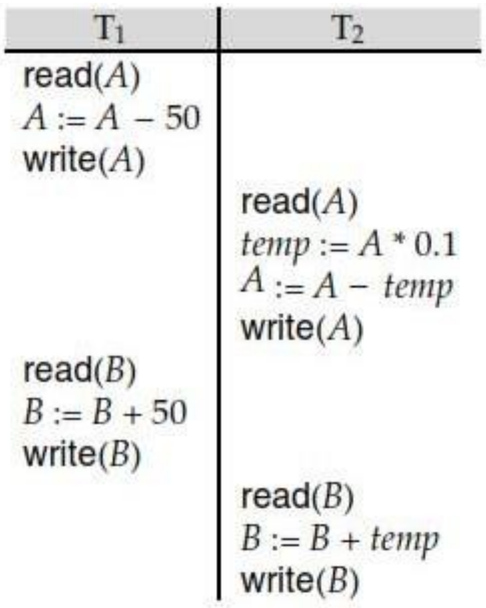
Ex: In transaction T1, the instruction write(A) must appear before the instruction read(B), in any valid schedule. In the following discussion, we shall refer to the first execution sequence (T1 followed by T2) as schedule 1, and to the second execution sequence (T2 followed by T1)as schedule2.

These schedules are serial: Each serial schedule consists of a sequence of instructions from various transactions, where the instructions belonging to one single transaction appear together in that schedule. Thus, for a set of n transactions, there exist n! different valid serial schedules.

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.

In general, it is not possible to predict exactly how many instructions of a transaction will be executed before the CPU switches to another transaction. Thus, the number of possible schedules for a set of n transactions is much larger than n!.

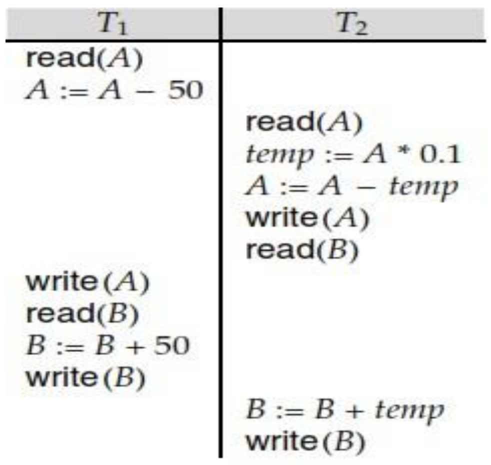
One possible schedule appears in Figure below. After this execution takes place, we arrive at the same state as the one in which the transactions are executed serially in the order T1 followed by T2. The sum A + B is indeed preserved.



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

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)

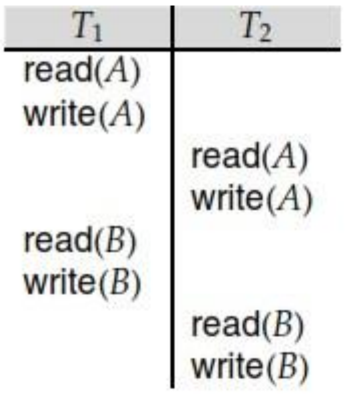
Schedule 4—a concurrent schedule.

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

The database system must control concurrent execution of transactions, to ensure that the database state remains consistent. Before we examine how the database system can carry out this task, we must first understand which schedules will ensure consistency, and which schedules will not. Since transactions are programs, it is computationally difficult to determine exactly what operations a transaction performs and how operations of various transactions interact. For this reason, we shall not interpret the type of operations that a transaction can perform on a data item. Instead, we consider only two operations: read and write. We thus assume that, between a read (Q) instruction and a write (Q) instruction on a data item Q, a transaction may perform an arbitrary sequence of operations on the copy of Q that is residing in the local buffer of the transaction. Thus, the only significant operations of a transaction, from a scheduling point of view, are its read and write instructions. We shall therefore usually show only read and write instructions in schedules, as we do in schedule 3 in Figure below.



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

Schedule 3—showing only the read and write instructions.

Two forms of serializabilty are

1. Conflict Serializabilty
2. 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:

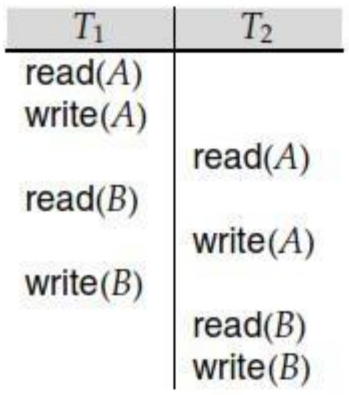
1. Ii = read(Q), Ij = read(Q). The order of Ii and Ij does not matter, since the same value of Q is read by Ti and Tj, regardless of the order.
2. Ii= read(Q), Ij= write(Q).If Ii comes before Ij, then Ti does not read the value of Q that is written by Tj in instruction Ij. If Ij comes before Ii, then 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 T1 with the read(A) instruction of T2

.

- Swap the write(B) instruction of T1 with the write(A) instruction of T2

.

- Swap the write(B) instruction of T1 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

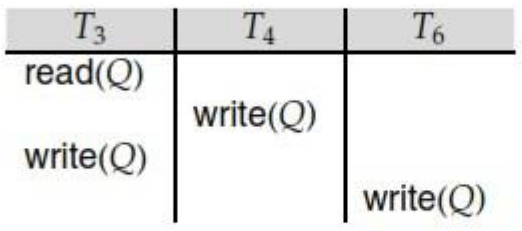
of transaction Ti must, in schedule S value of Q that was produced by the same write(Q) operation of transaction Tj.

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:



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

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 j such that T reads a data item previously written by Ti , the commit operation of Ti 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 fig below:

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).

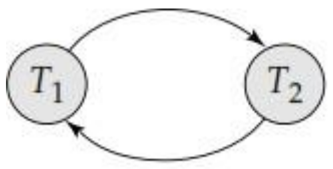
If an edge Ti → Tj exists in the precedence graph, then, in any serial schedule S' equivalent to S, must appear before Tj.

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.

T_1
T_2
(a)
T_2
T_1
(b)

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

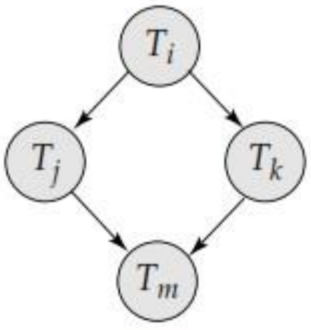
The precedence graph for schedule 4 appears in Figure below:



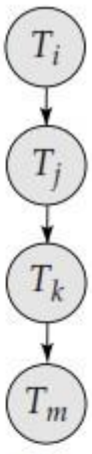
Precedence graph for schedule 4.

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.

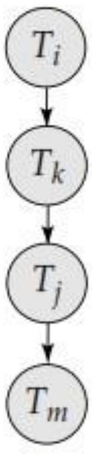
A serializability order of the transactions can be obtained through topological sorting, which determines a linear order consistent with the partial order of the precedence graph. There are, in general, several possible linear orders that can be obtained through a topological sorting. For example, the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

9. 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

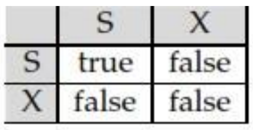
There are various modes in which a data item may be locked. In this section, we restrict our attention to two modes:

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.

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.

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 Ti requests a lock of mode A on item Q on which transaction Tj (Ti ≠ Tj) currently holds a lock of mode B. If transaction Ti 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. The compatibility relation between the two modes of locking discussed in this section appears in the matrix comp of Figure below:



	S	X
S	true	false
X	false	false

Lock-compatibility matrix comp.

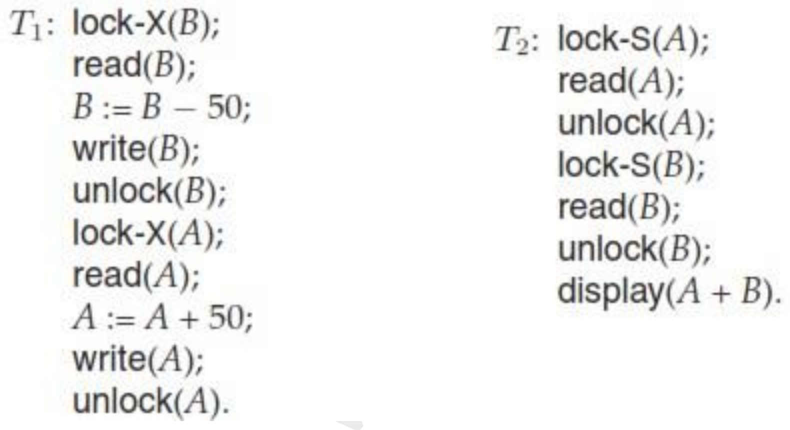
An element comp(A, B) of the matrix has the value true if and only if mode A is compatible with mode B. Note that shared mode is compatible with shared mode, but not with exclusive mode. At any time, several shared-mode locks can be held simultaneously (by different transactions) on a particular data item. A subsequent exclusive-mode lock request has to wait until the currently held shared-mode locks are released.

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.

To access a data item, transaction Ti must first lock that item. If the data item is already locked by another transaction in incompatible mode, the control manager will not grant the lock until all incompatible locks held by other transactions have been released. Thus, Ti is made to wait until all incompatible locks held by other transactions have been released.

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.

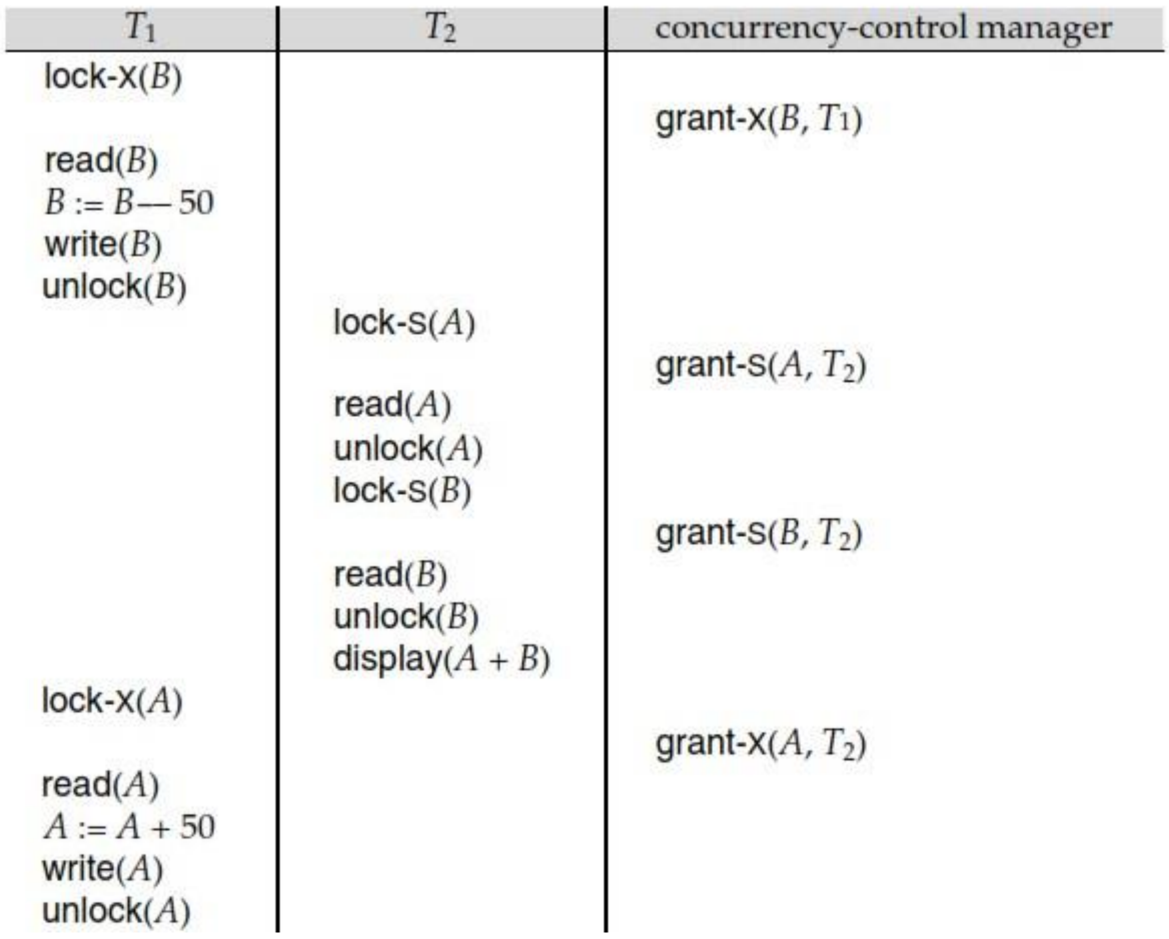
Ex: Let A and B be two accounts that are accessed by transactions T1 and T2.Transaction T1 transfers $50 from account B to account A. Transaction T2 displays the total amount of money in accounts A and B—that is, the sum A + B (transactions are shown in fig below).



```
T1: lock-X(B);
    read(B);
    B := B − 50;
    write(B);
    unlock(B);
    lock-X(A);
    read(A);
    A := A + 50;
    write(A);
    unlock(A).
```

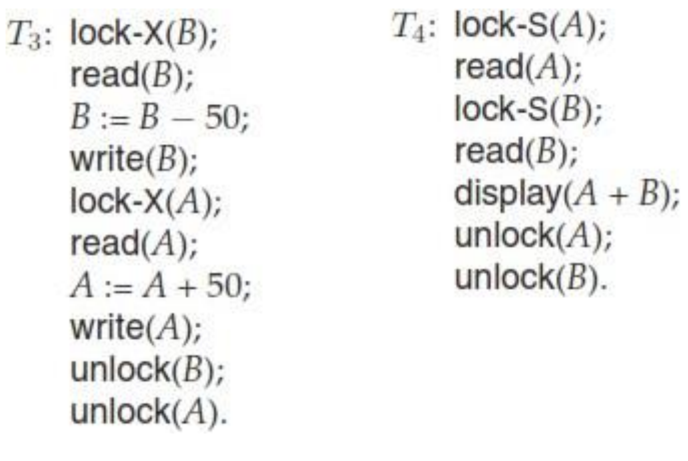
```
T2: lock-S(A);
    read(A);
    unlock(A);
    lock-S(B);
    read(B);
    unlock(B);
    display(A + B).
```

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.



T_1	T_2	concurrency-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B)		
$B := B — 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A, T_2)
read(A)		
$A := A + 50$		
write(A)		
unlock(A)		

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);            T4: lock-S(A);
    read(B);                  read(A);
    B := B − 50;              lock-S(B);
    write(B);                 read(B);
    lock-X(A);                display(A + B);
    read(A);                  unlock(A);
    A := A + 50;              unlock(B).
    write(A);
    unlock(B);
    unlock(A).
```

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.

Let { T0,T1,…..Tn} be a set of transactions participating in a schedule S. We say that Ti precedes Tj in S, written Ti → Tj has held lock mode A on Q, and Ti has held lock mode B on Q later, ,and Tj has held lock mode B on Q later, and comp(A,B)=false. If Ti → Tj, then that precedence implies that in any equivalent serial schedule, must appear before Tj.

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

When a transaction requests a lock on a data item in a particular mode, and no other transaction has a lock on the same data item in a conflicting mode, the lock can be granted. However, care must be taken to avoid the following scenario. Suppose a transaction T2 has a shared-mode lock on a data item, and another transaction T1 requests an exclusive-mode lock on the data item. Clearly, T1 has to wait for T2 to release the shared-mode lock. Meanwhile, a transaction T3 may request a shared-mode lock on the same data item. The lock request is compatible with the lock granted to T2,so T3 may be granted the shared-mode lock. At this point T3 may release the lock, but still T1 has to wait for T2 to finish. But again, there may be a new transaction T4 that requests a shared-mode lock on the same data item, and is granted the lock before T4 releases it. In fact, it is possible that there is a sequence of transactions that each requests a shared-mode lock on the data item, and each transaction releases the lock a short while after it is granted, but T1 never gets the exclusive-mode lock on the data item. The transaction T1 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 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 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 T3 and T4 are two phase, but, in schedule 2 (Figure above), 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.

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 teo-phase locking protocol, then T8 must loak 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.

This observation leads us to a refinement of the basic two-phase locking protocol, in which lock conversions are allowed. We shall provide a mechanism for *upgrading* a shared lock to an exclusive lock, and *downgrading* an exclusive lock to a shared lock. We denote conversion from shared to exclusive modes by upgrade, and from exclusive to shared by downgrade. Lock conversion cannot be allowed arbitrarily. Rather, upgrading can take place in only the growing phase, whereas downgrading can take place in only the shrinking phase.

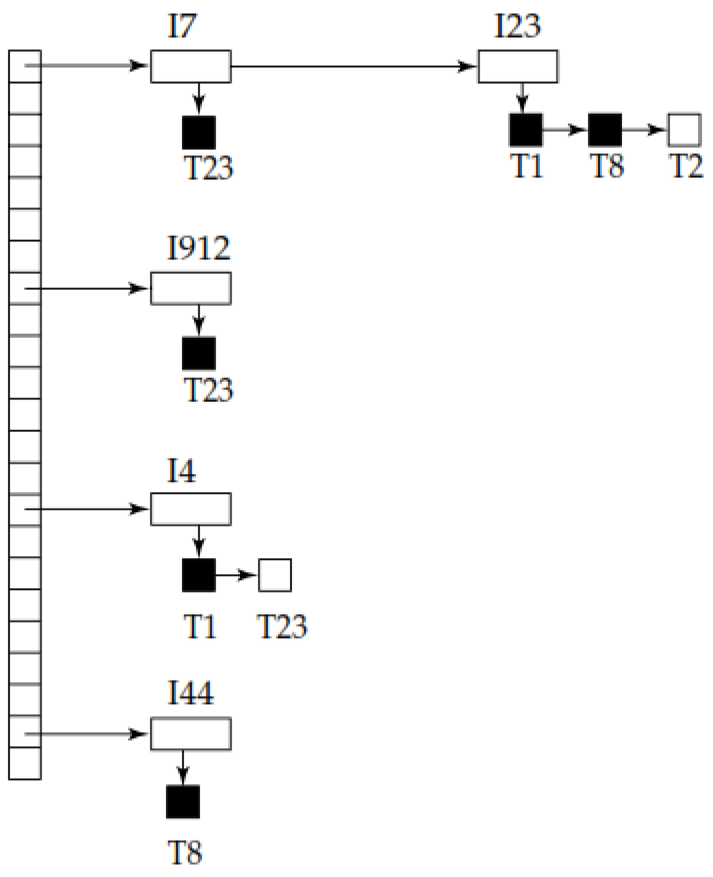
Strict two-phase locking and rigorous two-phase locking (with lock conversions) are used extensively in commercial database systems. A simple but widely used scheme automatically generates the appropriate lock and unlock instructions for a transaction, on the basis of read and write requests from the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

Implementation of Locking

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.

The lock manager uses this data structure: For each data item that is currently locked, it maintains a linked list of records, one for each request, in the order in which the requests arrived. It uses a hash table, indexed on the name of a data item, to find the linked list (if any) for a data item; this table is called the lock table. Each record of the linked list for a data item notes which transaction made the request, and what lock mode it requested. The record also notes if the request has currently been granted. The following fig shows example of a lock table.



The table contains locks for five different data items, I4, I7, I23, I44, and I912. The lock table uses overflow chaining, so there is a linked list of data items for each entry in the lock table. There is also a list of transactions that have been granted locks, or are waiting for locks, for each of the data items. Granted locks are the filled-in (black) rectangles, while waiting requests are the empty rectangles. We have omitted the lock mode to keep the figure simple. It can be seen, for example, that T23 has been granted locks on I912 and I7, and is waiting for a lock on I4. Although the figure does not show it, the lock table should also maintain an index on transaction identifiers, so that it is possible to determine efficiently the set of locks held by a given transaction.

The lock manager processes requests this way:

- When a lock request message arrives, it adds a record to the end of the linked list for the data item, if the linked list is present. Otherwise it creates a new linked list, containing only the record for the request. It always grants the first lock request on a data item. But if the transaction requests a lock on an item on which a lock has already been granted, the lock manager grants the request only if it is compatible with all earlier requests, and all earlier requests have been granted already. Otherwise the request has to wait.
- When the lock manager receives an unlock message from a transaction, it deletes the record for that data item in the linked list corresponding to that transaction. It tests the record that follows, if any, as described in the previous paragraph, to see if that

request can now be granted. If it can, the lock manager grants that request, and processes the record following it, if any, similarly, and so on.

- If a transaction aborts, the lock manager deletes any waiting request made by the transaction. Once the database system has taken appropriate actions to undo the transaction, it releases all locks held by the aborted transaction.

This algorithm guarantees freedom from starvation for lock requests, since a request can never be granted while a request received earlier is waiting to be granted.

Graph-Based Protocols

The two-phase locking protocol is both necessary and sufficient for ensuring serializability in the absence of information concerning the manner in which data items are accessed. But, if we wish to develop protocols that are not two phase, we need additional information on how each transaction will access the database. There are various models that can give us the additional information, each differing in the amount of information provided. The simplest model requires that we have prior knowledge about the order in which the database items will be accessed. Given such information, it is possible to construct locking protocols that are not two phase, but that, nevertheless, ensure conflict serializability.

To acquire such prior knowledge, we impose a partial ordering → on the set D = {d1,d2.,….. dn} of all data items. If di → dj , then any transaction accessing both di and dj must access di before accessing dj. This partial ordering may be the result of either the logical or the physical organization of the data, or it may be imposed solely for the purpose of concurrency control.

The partial ordering implies that the set D may now be viewed as a directed acyclic graph, called a database graph. For the sake of simplicity, we will restrict our attention to only those graphs that are rooted trees. We will present a simple protocol, called the tree protocol, which is restricted to employ only exclusive locks. In the tree protocol, the only lock instruction allowed is lock-X. Each transaction Ti can lock a data item at most once, and must observe the following rules:

1. The first lock by Ti may be on any data item.

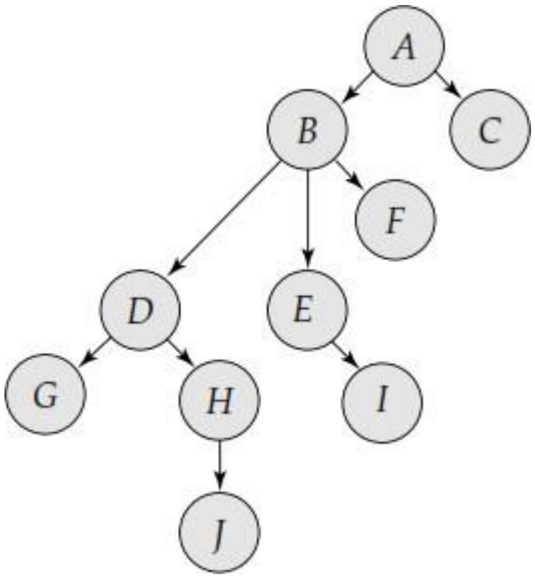
2. Subsequently, a data item Q can be locked by Ti only if the parent of Q is currently locked by Ti.

3. Data items may be unlocked at any time.

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

All schedules that are legal under the tree protocol are conflict serializable.

To illustrate this protocol, consider the database graph of Figure below:



The following four transactions follow the tree protocol on this graph. We show only the lock and unlock instructions:

T_{10}: lock-X(B); lock-X(E); lock-X(D); unlock(B); unlock(E); lock-X(G); unlock(D); unlock(G).
T_{11}: lock-X(D); lock-X(H); unlock(D); unlock(H).
T_{12}: lock-X(B); lock-X(E); unlock(E); unlock(B).
T_{13}: lock-X(D); lock-X(H); unlock(D); unlock(H).

One possible schedule in which these four transactions participated appears Figure below. Note that, during its execution, transaction T10 holds locks on two disjoint subtrees.

T_{10}	T_{11}	T_{12}	T_{13}
lock-X(B)			
	lock-X(D)		
	lock-X(H)		
	unlock(D)		
lock-X(E)			
lock-X(D)			
unlock(B)			
unlock(E)			
		lock-X(B)	
		lock-X(E)	
	unlock(H)		
lock-X(G)			
unlock(D)			
			lock-X(D)
			lock-X(H)
			unlock(D)
			unlock(H)
		unlock(E)	
		unlock(B)	
unlock (G)			

Observe that the schedule of Figure above is conflict serializable. It can be shown not only that the tree protocol ensures conflict serializability, but also that this protocol ensures freedom from deadlock.

The tree protocol in Figure above does not ensure recoverability and cascadelessness. To ensure recoverability and cascadelessness, the protocol can be modified to not permit release of exclusive locks until the end of the transaction. Holding exclusive locks until the end of the transaction reduces concurrency. Here is an alternative that improves concurrency, but ensures only recoverability: For each data item with an uncommitted write we record which transaction performed the last write to the data item. Whenever a transaction Ti performs a read of an uncommitted data item, we record a commit dependency of Ti on the transaction that performed the last write to the data item. Transaction Ti is then not permitted to commit until the commit of all transactions on which it has a commit dependency. If any of these transactions aborts, Ti must also be aborted.

Advantages:

The tree-locking protocol has two advantages over the two-phase locking protocol

- It is deadlock-free, so no rollbacks are required.
- Unlocking may occur earlier. Earlier unlocking may lead to shorter waiting times, and to an increase in concurrency.

Disadvantages:

However, the protocol has the disadvantage that, in some cases,

- A transaction may have to lock data items that it does not access. For example, a transaction that needs.

10. 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.

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

```
T14: read(B);
     read(A);
     display(A + B).
```

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

```
T15: 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-timestamp(Q). however, here in those case where TS(Ti)≥ R- timestamp(Q). we ignore the obsolete write.

11. Validation-Based Protocols

A concurrency-control scheme imposes overhead of code execution and possible delay of transactions. It may be better to use an alternative scheme that imposes less overhead. A difficulty in reducing the overhead is that we do not know in advance which transactions will be involved in a conflict. To gain that knowledge, we need a scheme for monitoring the system.

We assume that each transaction Ti executes in two or three different phases in its lifetime, depending on whether it is a read-only or an update transaction. The phases are, in order:

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

2. Validation phase. Transaction Ti performs a validation test to determine whether it can copy to the database the temporary local variables that hold the results of write operations without causing a violation of serializability.

3. Write phase. If transaction Ti succeeds in validation (step 2), then the system applies the actual updates to the database. Otherwise, the system rolls back Ti

Each transaction must go through the three phases in the order shown. However, all three phases of concurrently executing transactions can be interleaved. To perform the validation test, we need to know when the various phases of transactions Ti took place. We shall, therefore, associate three different timestamps with transaction Ti

1. Start(Ti), the time when Ti started its execution.

2. Validation(Ti),the time when Ti finished its read phase and started its validation phase.

3. Finish(Ti),the time when Ti finished its write phase.

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

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

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

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

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.

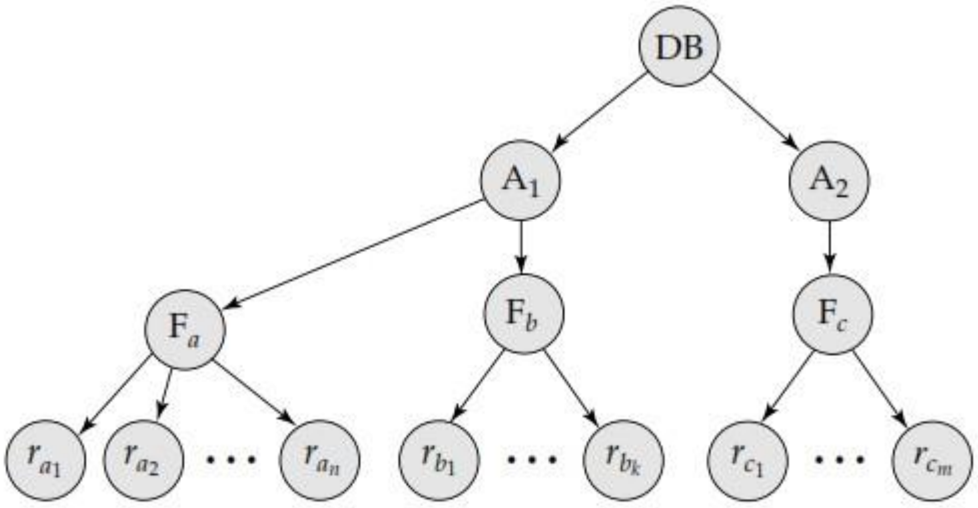
12. Multiple Granularity

In the concurrency-control schemes, 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.

Ex: if a transaction Ti needs to access the entire database, and a locking protocol is used, then Ti must lock each item in the database. Clearly, executing these locks is time consuming. It would be better if Ti could issue a single lock request to lock the entire database. On the other hand, if transaction Tj needs to access only a few data items, it should not be required to lock the entire database, since otherwise concurrency is lost.

This mechanism allows the system to define multiple levels of granularity. We can make one 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. Note that the tree that we describe here is significantly different from that used by the tree protocol. A nonleaf 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 below, which consists of four levels of nodes. The highest level represents the entire database. Below it are nodes of type area; the database consists of exactly these areas. Each area in turn has nodes of type file as its children. Each area contains exactly those files that are its child nodes. No file is in more than one area. Finally, each file has nodes of type record. As before, the file consists of exactly those records that are its child nodes, and no record can be present in more than one file.



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

Suppose that transaction Tj wishes to lock record rb6 of file Fb Since Ti has locked Fb explicitly, it follows that rb6 is also locked (implicitly). But, when Tj issues a lockrequest for rb6, rb6 is not explicitly locked! How does the system determine whether Tj can lock rb6? Tj must traverse the tree from the root to record rb6. If any node in that path is locked in an incompatible mode, then Tj must be delayed.

Suppose now that transaction Tk wishes to lock the entire database. To do so, it simply must lock the root of the hierarchy. Note, however, that Tk should not succeed in locking the root node, since Ti is currently holding a lock on part of the tree (specifically, on file F). But how does the system determine if the root node can be locked? One possibility is for it to search the entire tree. This solution, however, defeats the whole purpose of the multiple-granularity locking scheme.

A more efficient way to gain this knowledge is to introduce a new class of lock modes, called intention lock modes. If a node is locked in an intention mode, explicit locking is being 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. The compatibility function for these lock modes is in Figure below:

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

The multiple-granularity locking protocol, which ensures serializability, is this:

Each transaction Ti can lock a node Q by following these rules:

1. It must observe the lock-compatibility function

2. It must lock the root of the tree first, and can lock it in any mode.

3. It can lock a node Q in S or IS mode only if it currently has the parent of Q locked in either

IX or IS mode.

4. It can lock a node Q in X, SIX,orIX mode only if it currently has the parent of Q locked in either IX or SIX mode.

5. It can lock a node only if it has not previously unlocked any node (that is, T is two phase).

6. It can unlock a node Q only if it currently has none of the children of Q locked.

Observe that the multiple-granularity protocol requires that locks be acquired in topdown (root-to-leaf) order, whereas locks must be released in bottom-up (leaf-to-root)

order.

As an illustration of the protocol, consider the tree of Figure above and these transactions:

• Suppose that transaction T18 reads record ra2 in file Fa. Then, T18 needs to lock the database, area A1 ,and Fa in IS mode (and in that order), and finally to lock ra2 in S mode.

• Suppose that transaction T19 modifies record ra9 in file Fa. Then, T19 needs to lock the database, area A1,and file Fa in IX mode, and finally to lock ra2 in X mode.

• Suppose that transaction T20 reads all the records in file Fa. Then, T20 needs to lock the database and area A1(in that order) in IS mode, and finally to lock Fa in S mode.

• Suppose that transaction T21 reads the entire database. It can do so after locking the database in S mode.

We note that transactions T18, T20,and T21 can access the database concurrently. Transaction T19 can execute concurrently with T18, but not with either T20 or T21.

This protocol enhances concurrency and reduces lock overhead. It is particularly useful in applications that include a mix of

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

13. 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.

14. 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.

Deferred Database Modification

The deferred-modification technique ensures transaction atomicity by recording all database modifications in the log, but deferring the execution of all write operations of a transaction until the transaction partially commits. Recall that a transaction is said to be partially committed once the final action of the transaction has been executed. The version of the deferred-modification technique that we describe in this section assumes that transactions are executed serially.

When a transaction partially commits, the information on the log associated with the transaction is used in executing the deferred writes. If the system crashes before the transaction completes its execution, or if the transaction aborts, then the information on the log is simply ignored. The execution of transaction Ti proceeds as follows. Before Ti starts its execution, a record<Ti start> is written to the log. A write(X) operation by Ti results in the writing of a new record to the log. Finally, when <Ti commit> is written to the log.

When transaction Ti partially commits, a record Ti partially commits, the records associated with it in the log are used in executing the deferred writes. Since a failure may occur while this updating is taking place, we must ensure that, before the start of these updates, all the log records are written out to stable storage. Once they have been written, the actual updating takes place, and the transaction enters the committed state. Observe that only the new value of the data item is required by the deferred modification technique. Thus, we can simplify the general update-log record structure that we saw in the previous section, by omitting the old-value field.

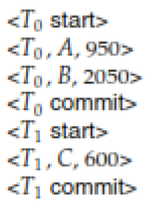
To illustrate, reconsider our simplified banking system. Let T0 be a transaction that transfers $50 from account A to account B:

```
T0: read(A);
    A := A - 50;
    write(A);
    read(B);
    B := B + 50;
    write(B).
```

Let T1 be a transaction that withdraws $100 from account C:

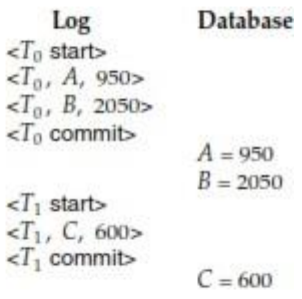
```
T1: read(C);
    C := C - 100;
    write(C).
```

Suppose that these transactions are executed serially, in the order T0 followed by T1 and that the values of accounts A, B, and C before the execution took place were $1000, $2000, and $700, respectively. The portion of the log containing the relevant information on these two transactions appears in Figure below:



<T_0 start>
<T_0, A, 950>
<T_0, B, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, 600>
<T_1 commit>

There are various orders in which the actual outputs can take place to both the database system and the log as a result of the execution of T0 and T1 .One such order appears in Figure below:



Log	Database
<T_0 start>	
<T_0, A, 950>	
<T_0, B, 2050>	
<T_0 commit>	
	A = 950
	B = 2050
<T_1 start>	
<T_1, C, 600>	
<T_1 commit>	
	C = 600

Note that the value of A is changed in the database only after the record <T,A,950> has been placed in the log. Using the log, the system can handle any failure that results in the loss of information on volatile storage. The recovery scheme uses the following recovery procedure:

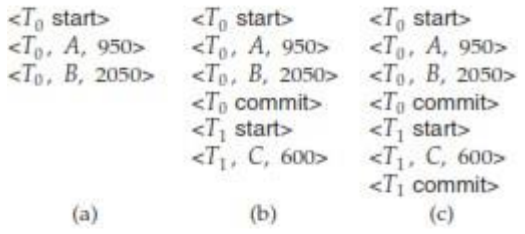
• redo(Ti) sets the value of all data items updated by transaction Ti to the new values.

The set of data items updated by Ti and their respective new values can be found in the log. The redo operation must be idempotent; that is, executing it several times must be equivalent to executing it once. This characteristic is required if we are to guarantee correct behavior even if a failure occurs during the recovery process. After a failure, the recovery subsystem consults the log to determine which transactions need to be redone. Transaction Ti needs to be redone if and only if the log contains both the record <Ti start> and the record <Ti commit>. Thus, if the system crashes after the transaction completes its execution, the recovery scheme uses the information in the log to restore the system to a previous consistent state after the transaction had completed.

As an illustration, let us return to our banking example with transactions T0 executed one after the other in the order T1 followed by T1. Figure the log that results from the complete execution of T0 and T1 . Let us suppose that the system crashes before the completion of the transactions, so that we can see how the recovery technique restores the database to a consistent state. Assume that the crash occurs just after the log record for the step of

write(B)

transaction T0 has been written to stable storage. The log at the time of the crash appears in Figure(a) below.



$<T_0$ start>	$<T_0$ start>	$<T_0$ start>
$<T_0, A, 950>$	$<T_0, A, 950>$	$<T_0, A, 950>$
$<T_0, B, 2050>$	$<T_0, B, 2050>$	$<T_0, B, 2050>$
	$<T_0$ commit>	$<T_0$ commit>
	$<T_1$ start>	$<T_1$ start>
	$<T_1, C, 600>$	$<T_1, C, 600>$
		$<T_1$ commit>
(a)	(b)	(c)

When the system comes back up, no redo actions need to be taken, since no commit record appears in the log. The values of accounts A and B remain $1000 and $2000, respectively. The log records of the incomplete transaction T0 can be deleted from the log.

Now, let us assume the crash comes just after the log record for the step of transaction T1

write(C)

has been written to stable storage. In this case, the log at the time of the crash is as in Figure(b). When the system comes back up, the operation redo(T0) is performed, since the record <T0 commit> appears in the log on the disk. After this operation is executed, the values of accounts A and B are $950 and $2050, respectively. The value of account C remains $700. As before, the log records of the incomplete transaction T1 can be deleted from the log.

Finally, assume that a crash occurs just after the log record <T1 commit> is written to stable storage. The log at the time of this crash is as in Figure(c). When the system comes back up, two commit records are in the log: one for T0 and one for T1 . Therefore, the system must perform operations redo(T0)and redo(T1) order in which their commit records appear in the log. After the system executes these operations, the values of accounts A, B, and C are $950, $2050, and $600, respectively. Finally, let us consider a case in which a second system crash occurs during recovery from the first crash. Some changes may have been made to the database as a result of the redo operations, but all changes may not have been made. When the system comes up after the second crash, recovery proceeds exactly as in the preceding examples. For each commit record <Ti commit> found in the log, the system performs the operation redo(Ti). In other words, it restarts the recovery actions from the beginning. Since redo writes values to the database independent of the values currently in the database, the result of a successful second attempt at redo is the same as though redo had succeeded the first time.

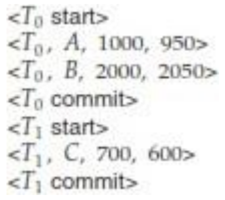
Immediate Database Modification

The immediate-modification technique allows database modifications to be output to the database while the transaction is still in the active state. Data modifications written by active transactions are called uncommitted modifications. In the event of a crash or a transaction failure, the system must use the old-value field of the log records to restore the modified data items to the value they had prior to the start of the transaction. The undo operation, described next, accomplishes this restoration.

Before a transaction <Ti start> starts its execution, the system writes the record <Ti start> to the log. During its execution, any write(X)operation by Ti is preceded by the writing of the appropriate new update record to the log. When Ti partially commits, the system writes the record <Ti commit> to the log.

Since the information in the log is used in reconstructing the state of the database, we cannot allow the actual update to the database to take place before the corresponding log record is written out to stable storage. We therefore require that, before execution of an output(B) operation, the log records corresponding to B be written onto stable storage.

As an illustration, let us reconsider our simplified banking system, with transactions T0 and T1 executed one after the other in the order T0 followed by T1. The portion of the log containing the relevant information concerning these two transactions appears in Figure below:



$<T_0$ start$>$
$<T_0, A, 1000, 950>$
$<T_0, B, 2000, 2050>$
$<T_0$ commit$>$
$<T_1$ start$>$
$<T_1, C, 700, 600>$
$<T_1$ commit$>$

The following Figure shows one possible order in which the actual outputs took place in both the database system and the log as a result of the execution of T0 and T1. Notice that this order could not be obtained in the deferred-modification technique.

Using the log, the system can handle any failure that does not result in the loss of information in nonvolatile storage. The recovery scheme uses two recovery procedures:

• undo(Ti) restores the value of all data items updated by transaction Ti to the old values.

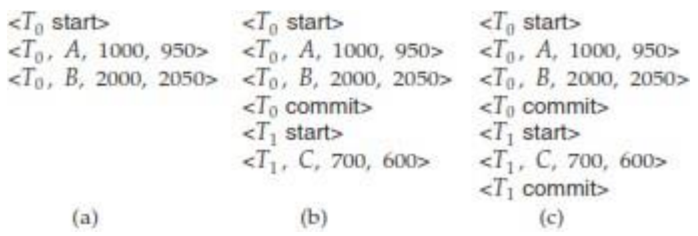
• redo(Ti) sets the value of all data items updated by transaction Ti to the new values.

The set of data items updated by Ti and their respective old and new values can be found in the log. The undo and redo operations must be idempotent to guarantee correct behaviour even if a failure occurs during the recovery process. After a failure has occurred, the recovery scheme consults the log to determine which transactions need to be redone, and which need to be undone:

• Transaction Ti needs to be undone if the log contains the record <Ti start>, but does not contain the record <Ti commit>.

• Transaction Ti needs to be redone if the log contains both the record <Ti start> and the record <Ti commit>.

As an illustration, return to our banking example, with transaction T0 and T1 executed one after the other in the order T0 followed by T1. Suppose that the system crashes before the completion of the transactions. We shall consider three cases. The state of the logs for each of these cases appears in Figure below:



(a)	(b)	(c)
$<T_0$ start>	$<T_0$ start>	$<T_0$ start>
$<T_0, A, 1000, 950>$	$<T_0, A, 1000, 950>$	$<T_0, A, 1000, 950>$
$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$
	$<T_0$ commit>	$<T_0$ commit>
	$<T_1$ start>	$<T_1$ start>
	$<T_1, C, 700, 600>$	$<T_1, C, 700, 600>$
		$<T_1$ commit>

First, let us assume that the crash occurs just after the log record for the step of transaction T0 has been written to stable storage(fig a)

write(B)

When the system comes back up two recovery actions need to be taken. The operation undo(T1)must be performed, since the record <T1 start> appears in the log, but there is no record <T1 commit>.The operation redo(T1) must be performed, since the log contains both the record <T0 start> and the record <T0 commit>. At the end of the entire recovery procedure, the values of accounts A, B,andC are $950, $2050, and $700, respectively. Note that the undo(T1) operation is performed before the redo(T0). In this example, the same outcome would result if the order were reversed. However, the order of doing undo operations first, and then redo operations, is important for the recovery algorithm.

Finally, let us assume that the crash occurs just after the log record

<T1 commit>

has been written to stable storage (Figure c). When the system comes back up, both T0 and T1 need to be redone, since the records <T0 start> and <T1 commit> appear in the log, as do the records <T1 start> and <T1 commit>. After the system performs the recovery procedures redo(T0)and redo(T1), the values in accounts A, B, and C are $950, $2050, and $600, respectively.

Checkpoints

When a system failure occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone. In principle, we need to search the entire log to determine this information. There are two major difficulties with this approach:

1. The search process is time consuming.

2. Most of the transactions that, according to our algorithm, need to be redone have already written their updates into the database. Although redoing them will cause no harm, it will nevertheless cause recovery to take longer.

To reduce these types of overhead, we introduce checkpoints. During execution, the system maintains the log. In addition, the system periodically performs checkpoints, which require the following sequence of actions to take place:

1. Output onto stable storage all log records currently residing in main memory.

2. Output to the disk all modified buffer blocks.

3. Output onto stable storage a log record <checkpoint>.

Transactions are not allowed to perform any update actions, such as writing to a buffer block or writing a log record, while a checkpoint is in progress. The presence of a <checkpoint> record in the log allows the system to streamline its recovery procedure. Consider a transaction Ti that committed prior to the checkpoint.

For such a transaction, the <Ti commit> record appears in the log before the <checkpoint> record. Any database modification made by Ti must have been written to the database either prior to the checkpoint or as part of the checkpoint itself. Thus, at recovery time, there is no need to perform a redo operation on Ti.

This observation allows us to refine our previous recovery schemes. After a failure has occurred, the recovery scheme examines the log to determine the most recent transaction Ti that started executing before the most recent checkpoint took place. It can find such a transaction by searching the log backward, from the end of the log, until it finds the first <checkpoint> record (since we are searching backward, the record found is the final <checkpoint> record in the log); then it continues the search backward until it finds the next <Ti start> record. This record identifies a transaction Ti.

Once the system has identified transaction Ti, the redo and undo operations needto be applied to only transaction Ti and all transactions Tj that started executing after transaction Ti. Let us denote these transactions by the set T. The remainder (earlier part) of the log can be ignored, and can be erased whenever desired. The exact recovery operations to be performed depend on the modification technique being used. For the immediate-modification technique, the recovery operations are:

• For all transactions Tk in T that have no <Ti commit> record in the log, execute undo(Tk)

• For all transactions Tk in T that have no <Tk commit> appears in the log, execute redo(Tk).

Obviously, the undo operation does not need to be applied when the deferred-modification technique is being employed.

As an illustration, consider the set of transactions {T0, T1,…….T100} executed in the order of the subscripts. Suppose that the most recent checkpoint took place during the execution of transaction T67. Thus, only transactions T67, T68,………..T100 need to be considered during the recovery scheme. Each of them needs to be redone if it has committed; otherwise, it needs to be undone.

15. 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.

Transaction Rollback

We roll back a failed transaction, Ti, by using the log. The system scans the log backward; for every log record of the form <Ti, Xj, V1,V2> found in the log, the system restores the data item Xj to its old value V1. Scanning of the log terminates when the log record <Ti , start> is found. Scanning the log backward is important, since a transaction may have updated a data item more than once. As an illustration, consider the pair of log records

<T,A,10, 20>

<Ti,A,20, 30>

The log records represent a modification of data item A by Ti, followed by another modification of A by Ti. Scanning the log backward sets A correctly to 10.Ifthelog were scanned in the forward direction, A would be set to 20, which is incorrect. If strict two-phase locking is used for concurrency control, locks held by a transaction T may be released only after the transaction has been rolled back as described.

Once transaction T (that is being rolled back) has updated a data item, no other transaction could have updated the same data item, because of the concurrency-control requirements. Therefore, restoring the old value of the data item will not erase the effects of any other transaction.

Checkpoints

we used checkpoints to reduce the number of log records that the system must scan when it recovers from a crash. Since we assumed no concurrency, it was necessary to consider only the following transactions during recovery:

• Those transactions that started after the most recent checkpoint

• The one transaction, if any, that was active at the time of the most recent checkpoint.

The situation is more complex when transactions can execute concurrently, since several transactions may have been active at the time of the most recent checkpoint.

In a concurrent transaction-processing system, we require that the checkpoint log record be of the form <checkpoint L>,where L is a list of transactions active at the time of the checkpoint. Again, we assume that transactions do not perform updates either on the buffer blocks or on the log while the checkpoint is in progress. The requirement that transactions must not perform any updates to buffer blocks or to the log during check pointing can be bothersome, since transaction processing will have to halt while a checkpoint is in progress. A fuzzy checkpoint is a checkpoint where transactions are allowed to perform updates even while buffer blocks are being written out.

Restart Recovery

When the system recovers from a crash, it constructs two lists: The undo-list consists of transactions to be undone, and the redo-list consists of transactions to be redone. The system constructs the two lists as follows: Initially, they are both empty. The system scans the log backward, examining each record, until it finds the first <checkpoint> record:

• For each record found of the form <Ti commit>, it adds Ti to redo-list.

• For each record found of the form <Ti start>,if Ti is not in redo-list, then it adds Ti to undo-list.

When the system has examined all the appropriate log records, it checks the list L in the checkpoint record. For each transaction Ti in L, if Ti is not in redo-list then it adds Ti to the undo-list.

Once the redo-list and undo-list have been constructed, the recovery proceeds as follows:

1. The system rescans the log from the most recent record backward, and performs an undo for each log record that belongs transaction Ti on the undo-list. Log records of transactions on the redo-list are ignored in this phase. The scan stops when the <Ti start> records have been found for every transaction Ti in the undo-list.
2. The system locates the most recent <checkpoint L> record on the log. Notice that this step may involve scanning the log forward, if the checkpoint record was passed in step 1.
3. The system scans the log forward from the most recent <checkpoint L>record, and performs redo for each log record that belongs to a transaction Ti that is on the redo-list. It ignores log records of transactions on the undo-list in this phase.

It is important in step 1 to process the log backward, to ensure that the resulting state of the database is correct.

After the system has undone all transactions on the undo-list, it redoes those transactions on the redo-list. It is important, in this case, to process the log forward. When the recovery process has completed, transaction processing resumes. It is important to undo the transaction in the undo-list before redoing transactions in the redo-list, using the algorithm in steps 1 to 3; otherwise, a problem may occur. Suppose that data item A initially has the value 10. Suppose that a transaction Ti updated data item A to 20 and aborted; transaction rollback would restore A to the value 10. Suppose that another transaction Ti then updated data item A to 30 and committed, following which the system crashed. The state of the log at the time of the crash is

<Ti, A, 10, 20>

<Tj, A, 10, 30>

<Tj commit>

If the redo pass is performed first, A will be set to 30; then, in the undo pass, A will be set to 10, which is wrong. The final value of Q should be 30, which we can ensure by performing undo before performing redo.

Buffer Management

Now we consider several subtle details that are essential to the implementation of a crash-recovery scheme that ensures data consistency and imposes a minimal amount of overhead on interactions with the database.

Log-Record Buffering

So far, we have assumed that every log record is output to stable storage at the time it is created. This assumption imposes a high overhead on system execution for several reasons: Typically, output to stable storage is in units of blocks. In most cases, a log record is much smaller than a block. Thus, the output of each log record translates to a much larger output at the physical level. The output of a block to stable storage may involve several output operations at the physical level.

The cost of performing the output of a block to stable storage is sufficiently high that it is desirable to output multiple log records at once. To do so, we write log records to a log buffer in main memory, where they stay temporarily until they are output to stable storage. Multiple log records can be gathered in the log buffer, and output to stable storage in a single output operation. The order of log records in the stable storage must be exactly the same as the order in which they were written to the log buffer.

As a result of log buffering, a log record may reside in only main memory (volatile storage) for a considerable time before it is output to stable storage. Since such log records are lost if the system crashes, we must impose additional requirements on the recovery techniques to ensure transaction atomicity:

• Transaction Ti enters the commit state after the <Ti commit> log record has been output to stable storage.

• Before the <Ti commit> log record can be output to stable storage, all log records pertaining to transaction Ti must have been output to stable storage.

• Before a block of data in main memory can be output to the database (in non-volatile storage), all log records pertaining to data in that block must have been output to stable storage.

This rule is called the write-ahead logging (WAL) rule. The three rules state situations in which certain log records must have been output to stable storage. There is no problem resulting from the output of log records earlier than necessary. Thus, when the system finds it necessary to output a log record to stable storage, it outputs an entire block of log records, if there are enough log records in main memory to fill a block. If there are insufficient log records to fill the block, all log records in main memory are combined into a partially full block, and are output to stable storage. Writing the buffered log to disk is sometimes referred to as a log force.

Database Buffering

In Log-Record buffering, we described the use of a two-level storage hierarchy. The system stores the database in non-volatile storage (disk), and brings blocks of data into main memory as needed. Since main memory is typically much smaller than the entire database, it may be necessary to overwrite a block B in main memory when another block B2 needs to be brought into memory. If B1 has been modified B1 must be output prior to the input of B2, this storage hierarchy is the standard operating system concept of virtual memory. The rules for the output of log records limit the freedom of the system to output blocks of data. If the input of block B2 causes block B1 to be chosen for output, all log records pertaining to data in B1 must be output to stable storage before B1 is output. Thus, the sequence of actions by the system would be:

• Output log records to stable storage until all log records pertaining to block B1 have been output.

• Output block B1

• Input blocks B2 from disk to main memory.

It is important that no writes to the block B1be in progress while the system carries out this sequence of actions. We can ensure that there are no writes in progress by using a special means of locking: Before a transaction performs a write on a data item, it must acquire an exclusive lock on the block in which the data item resides.

The lock can be released immediately after the update has been performed. Before a block is output, the system obtains an exclusive lock on the block, to ensure that no transaction is updating the block. It releases the lock once the block output has completed. Locks that are held for a short duration are often called latches. Latches are treated as distinct from locks used by the concurrency-control system. As a result, they may be released without regard to any locking protocol, such as two-phase locking, required by the concurrency-control system.

To illustrate the need for the write-ahead logging requirement, consider our banking example with transactions T0 and T1:

. Suppose that the state of the log is <T0 start>

<T0,A,1000, 950>

and that transaction T0 issues a read(B). Assume that the block on which B resides is not in main memory, and that main memory is full. Suppose that the block on which A resides is chosen to be output to disk. If the system outputs this block to disk and then a crash occurs, the values in the database for accounts A, B,andC are $950, $2000, and $700, respectively. This database state is inconsistent. However, because of the WAL requirements, the log record <T0 ,A,1000, 950> must be output to stable storage prior to output of the block on which A resides. The system can use the log record during recovery to bring the database back to a consistent state.

Operating System Role in Buffer Management

We can manage the database buffer by using one of two approaches:

1. The database system reserves part of main memory to serve as a buffer that it, rather than the operating system, manages. The database system manages data-block transfer in accordance with the requirements. This approach has the drawback of limiting flexibility in the use of main memory. The buffer must be kept small enough that other applications have sufficient main memory available for their needs. However, even when the other applications are not running, the database will not be able to make use of all the available memory. Likewise, non-database applications may not use that part of main memory reserved for the database buffer, even if some of the pages in the database buffer are not being used.

2. The database system implements its buffer within the virtual memory provided by the operating system. Since the operating system knows about the memory requirements of all processes in the system, ideally it should be in charge of deciding what buffer blocks must be force-output to disk, and when. But, to ensure the write-ahead logging requirements, the operating system should not write out the database buffer pages itself, but in- stead should request the database system to force-output the buffer blocks.

The database system in turn would force-output the buffer blocks to the database, after writing relevant log records to stable storage. Unfortunately, almost all current-generation operating systems retain complete control of virtual memory. The operating system reserves space on disk for storing virtual-memory pages that are not currently in main memory; this space is called swap space. If the operating system decides to output a block Bx, that block is output to the swap space on disk, and there is no way for the database system to get control of the output of buffer blocks.

Therefore, if the database buffer is in virtual memory, transfers between database files and the buffer in virtual memory must be managed by the database system, which enforces the write-ahead logging requirements.

This approach may result in extra output of data to disk. If a block Bx is output by the operating system, that block is not output to the database. Instead, it is output to the swap space for the operating system's virtual memory. When the database system needs to output Bx, the operating system may need first to input Bx from its swap space. Thus, instead of a single output of Bx, there may be two outputs of Bx(one by the operating system, and one by the database system) and one extra input of Bx.

Although both approaches suffer from some drawbacks, one or the other must be chosen unless the operating system is designed to support the requirements of database logging. Only a few current operating systems, such as the Mach operating system, support these requirements.

Failure with Loss of Nonvolatile Storage

The basic scheme is to dump the entire content of the database to stable storage periodically—say, once per day. For example, we may dump the database to one or more magnetic tapes. If a failure occurs that results in the loss of physical database blocks, the system uses the most recent dump in restoring the database to a previous consistent state. Once this restoration has been accomplished, the system uses the log to bring the database system to the most recent consistent state. More precisely, no transaction may be active during the dump procedure, and a procedure similar to check pointing must take place:

1. Output all log records currently residing in main memory onto stable storage.

2. Output all buffer blocks onto the disk.

3. Copy the contents of the database to stable storage.

4. Output a log record <dump> onto the stable storage.

Steps 1, 2, and 4 correspond to the three steps used for checkpoints, To recover from the loss of nonvolatile storage, the system restores the database to disk by using the most recent dump. Then, it consults the log and redoes all the transactions that have committed since the most recent dump occurred. Notice that no undo operations need to be executed. A dump of the database contents is also referred to as an archival dump, since we can archive the dumps and use them later to examine old states of the database. Dumps of a database and check pointing of buffers are similar.

The simple dump procedure described here is costly for the following two reasons.:

- The entire database must be be copied to stable storage, resulting in considerable data transfer.

- Since transaction processing is halted during the dump procedure, CPU cycles are wasted. Fuzzy dump schemes have been developed, which allow transactions to be active while the dump is in progress. They are similar to fuzzy check pointing schemes; see the bibliographical notes for more details.

ARIES

The state of the art in recovery methods is best illustrated by the ARIES recovery method. The advanced recovery technique which we have described is modeled after ARIES, but has been simplified significantly to bring out key concepts and make it easier to understand. In contrast, ARIES uses a number of techniques to reduce the time taken for recovery, and to reduce the overheads of checkpointing. In particular, ARIES is able to avoid redoing many logged operations that have already been applied and to reduce the amount of information logged. The price paid is greater complexity; the benefits are worth the price.

The major differences between ARIES and our advanced recovery algorithm are that ARIES:

1. Uses a log sequence number (LSN) to identify log records, and the use of LSNs in database pages to identify which operations have been applied to a database page.

2. Supports physiological redo operations, which are physical in that the affected page is physically identified, but can be logical within the page. For instance, the deletion of a record from a page may result in many other records in the page being shifted, if a slotted page structure is used. With physical redo logging, all bytes of the page affected by the shifting of records must be logged. With physiological logging, the deletion operation can be logged, resulting in a much smaller log record. Redo of the deletion operation would delete the record and shift other records as required.

3. Uses a dirty page table to minimize unnecessary redos during recovery. Dirty pages are those that have been updated in memory, and the disk version is not up-to-date.

4. Uses fuzzy check pointing scheme that only records information about dirty pages and associated information, and does not even require writing of dirty pages to disk. It flushes dirty pages in the background, continuously, instead of writing them during checkpoints.

Data Structures

Each log record in ARIES has a log sequence number (LSN) that uniquely identifies the record. The number is conceptually just a logical identifier whose value is greater for log records that occur later in the log. In practice, the LSN is generated in such a way that it can also be used to locate the log record on disk. Typically, ARIES splits a log into multiple log files, each of which has a file number. When a log file grows to some limit, ARIES appends further log records to a new log file; the new log file has a file number that is higher by 1 than the previous log file.

The LSN then consists of a file number and an offset within the file. Each page also maintainsan identifier called the PageLSN. Whenever an operation (whether physical or logical) occurs on a page, the operation stores the LSN of its log record in the PageLSN field of the page. During the redo phase of recovery, any log records with LSN less than or equal to the PageLSN of a page should notbe executed on the page, since their actions are already reflected on the page. In combination with a scheme for recording PageLSNs as part of checkpointing, ARIES can avoid even reading many pages for which logged operations are already reflected on disk. Thereby recovery time is reduced significantly.

The PageLSN is essential for ensuring idempotence in the presence of physiological redo operations, since reapplying a physiological redo that has already been applied to a page could cause incorrect changes to a page. Pages should not be flushed to disk while an update is in progress, since physiological operations cannot be redone on the partially updated state of the page on disk. Therefore, ARIES uses latches on buffer pages to prevent them from being written to disk while they are being updated. It releases the buffer page latch only after the update is completed, and the log record for the update has been written to the log.

Each log record also contains the LSN of the previous log record of the same transaction. This value, stored in the PrevLSN field, permits log records of a transaction to be fetched backward, without reading the whole log. There are special redo-only log records generated during transaction rollback, called compensation log records (CLRs) in ARIES. These serve the same purpose as the redo-only log records in our advanced recovery scheme. In addition CLRs serve the role of the operation-abort log records in our scheme. The CLRs have an extra field, called the UndoNextLSN.

Recovery Algorithm

ARIES recovers from a system crash in three passes.

• **Analysis pass:** This pass determines which transactions to undo, which pages were dirty at the time of the crash, and the LSN from which the redo pass should start.

• **Redo pass:** This pass starts from a position determined during analysis, and performs a redo, repeating history, to bring the database to a state it was in before the crash.

• **Undo pass:** This pass rolls back all transactions that were incomplete at the time of crash.

Analysis Pass: The analysis pass finds the last complete checkpoint log record, and reads in the DirtyPageTable from this record. It then sets RedoLSN to the minimum of the RecLSNs of the pages in the DirtyPageTable. If there are no dirty pages, it sets RedoLSN to the LSN of the checkpoint log record. The redo pass starts its scan of the log from RedoLSN. All the log records earlier than this point have already been applied to the database pages on disk. The analysis pass initially sets the list of transactions to be undone, undo-list, to the list of transactions in the checkpoint log record. The analysis pass also reads from the checkpoint log record the LSNs of the last log record for each transaction in undo-list. The analysis pass continues scanning forward from the checkpoint. Whenever it finds a log record for a transaction not in the undo-list, it adds the transaction to undo-list. Whenever it finds a transaction end log record, it deletes the transaction from undo-list. All transactions left in undo-list at the end of analysis have to be rolled back later, in the undo pass. The analysis pass also keeps track of the last record of each transaction in undo-list, which is used in the undo pass.

The analysis pass also updates Dirty Page Table whenever it finds a log record for an update on a page. If the page is not in Dirty Page Table, the analysis pass adds it to Dirty Page Table, and sets the RecLSN of the page to the LSN of the log record.

Redo Pass: The redo pass repeats history by replaying every action that is not already reflected in the page on disk. The redo pass scans the log forward from RedoLSN. Whenever it finds an update log record, it takes this action:

1. If the page is not in Dirty Page Table or the LSN of the update log record is less than the RecLSN of the page in Dirty Page Table, then the redo pass skips the log record.

2. Otherwise the redo pass fetches the page from disk, and if the PageLSN is less than the LSN of the log record, it redoes the log record.

Note that if either of the tests is negative, then the effects of the log record have already appeared on the page. If the first test is negative, it is not even necessary to fetch the page from disk.

Undo Pass and Transaction Rollback: The undo pass is relatively straightforward. It performs a backward scan of the log, undoing all transactions in undo-list. If a CLR is found, it uses the UndoNextLSN field to skip log records that have already been rolled back. Otherwise, it uses the PrevLSN field of the log record to find the next log record to be undone.

Whenever an update log record is used to perform an undo (whether for transaction rollback during normal processing, or during the restart undo pass), the undo pass generates a CLR containing the undo action performed (which must be physiological). It sets the UndoNextLSN of the CLR to the PrevLSN value of the update log record.

Other Features

Among other key features that ARIES provides are:

• **Recovery independence:** Some pages can be recovered independently from others, so that they can be used even while other pages are being recovered. If some pages of a disk fail, they can be recovered without stopping transaction processing on other pages.

• **Save points:** Transactions can record save points, and can be rolled back partially, up to a savepoint. This can be quite useful for deadlock handling, since transactions can be rolled back up to a point that permits release of required locks, and then restarted from that point.

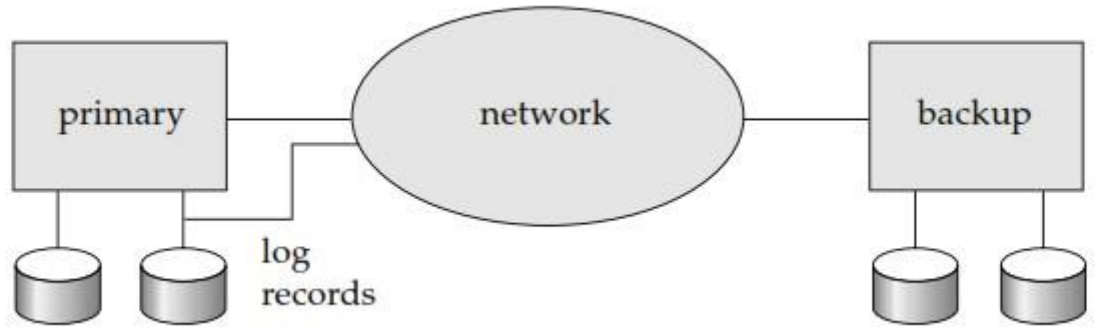
• **Fine-grained locking:** The ARIES recovery algorithm can be used with index concurrency control algorithms that permit tuple level locking on indices, instead of page level locking, which improves concurrency significantly.

• **Recovery optimizations:** The Dirty Page Table can be used to prefetch pages during redo, instead of fetching a page only when the system finds a log record to be applied to the page. Out-of-order redo is also possible: Redo can be postponed on a page being fetched from disk, and performed when the page is fetched. Meanwhile, other log records can continue to be processed.

6. Remote Backup Systems

Traditional transaction-processing systems are centralized or client–server systems. Such systems are vulnerable to environmental disasters such as fire, flooding, or earthquakes. Increasingly, there is a need for transaction-processing systems that can function in spite of system failures or environmental disasters. Such systems must provide high availability, that is, the time for which the system is unusable must be extremely small.

We can achieve high availability by performing transaction processing at one site, called the primary site, and having a remote backup site where all the data from the primary site are replicated. The remote backup site is sometimes also called the secondary site. The remote site must be kept synchronized with the primary site, as updates are performed at the primary. We achieve synchronization by sending all log records from primary site to the remote backup site. The remote backup site must be physically separated from the primary—for example, we can locate it in a different state—so that a disaster at the primary does not damage the remote backup site. Figure below shows the architecture of a remote backup system.



When the primary site fails, the remote backup site takes over processing. First, however, it performs recovery, using its (perhaps outdated) copy of the data from the primary, and the log records received from the primary. In effect, the remote backup site is performing recovery actions that would have been performed at the primary site when the latter recovered. Standard recovery algorithms, with minor modification, can be used for recovery at the remote backup site. Once recovery has been performed, the remote backup site starts processing transactions.

Availability is greatly increased over a single-site system, since the system can recover even if all data at the primary site are lost. The performance of a remote backup system is better than the performance of a distributed system with two-phase commit. Several issues must be addressed in designing a remote backup system:

• **Detection of failure**. As in failure-handling protocols for distributed system, it is important for the remote backup system to detect when the primary has failed. Failure of communication lines can fool the remote backup into believing that the primary has failed. To avoid this problem, we maintain several communication links with independent modes of failure between the primary and the remote backup. For example, in addition to the network connection, there may be a separate modem connection over a telephone line, with services provided by different telecommunication companies. These connections may be backed up via manual intervention by operators, who can communicate over the telephone system.

• **Transfer of control**. When the primary fails, the backup site takes over processing and becomes the new primary. When the original primary site recovers, it can either play the role of remote backup, or take over the role of primary site again. In either case, the old primary must receive a log of updates carried out by the backup site while the old primary was down. The simplest way of transferring control is for the old primary to receive redo logs from the old backup site, and to catch up with the updates by applying them locally. The old primary can then act as a remote backup site. If control must be transferred back, the old backup site can pretend to have failed, resulting in the old primary taking over.

• **Time to recover.** If the log at the remote backup grows large, recovery will take a long time. The remote backup site can periodically process the redo log records that it has received, and can perform a checkpoint, so that earlier parts of the log can be deleted. The delay before the remote backup takes over can be significantly reduced as a result. A hot-spare configuration can make takeover by the

• backup site almost instantaneous. In this configuration, the remote backup site continually processes redo log records as they arrive, applying the updates locally. As soon as the failure of the primary is detected, the backup site completes recovery by rolling back incomplete transactions; it is then ready to process new transactions.

• **Time to commit.** To ensure that the updates of a committed transaction are durable, a transaction must not be declared committed until its log records have reached the backup site. This delay can result in a longer wait to commit a transaction, and some systems therefore permit lower degrees of durability. The degrees of durability can be classified as follows.

One-safe: A transaction commits as soon as its commit log record is written to stable storage at the primary site. The problem with this scheme is that the updates of a committed transaction may not have made it to the backup site, when the backup site takes over processing. Thus, the updates may appear to be lost. When the primary site recovers, the lost updates cannot be merged in directly, since the updates may conflict with later updates performed at the backup site. Thus, human intervention may be required to bring the database to a consistent state.

Two-very-safe: A transaction commits as soon as its commit log record is written to stable storage at the primary and the backup site. The problem with this scheme is that transaction processing cannot proceed if either the primary or the backup site is down. Thus, availability is actually less than in the single-site case, although the probability of data loss is much less.

Two-safe: This scheme is the same as two-very-safe if both primary and backup sites are active. If only the primary is active, the transaction is allowed to commit as soon as its commit log record is written to stable storage at the primary site. This scheme provides better availability than does two-very-safe, while avoiding the problem of lost transactions faced by the one-safe scheme. It results in a slower commit than the one-safe scheme, but the benefits generally outweigh the cost.