UNIT-4

Syllabus:
Transaction Processing Concept: Transaction system, Testing of serializability, serializability of schedules, conflict & view serializable schedule, recoverability, Recovery from transaction failures, log-based recovery, checkpoints, deadlock handling

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

ACID Properties:
To ensure integrity of the data, we require that the database system maintain the following properties of the transactions known as ACID properties.

- **Atomicity**. Either all operations of the transaction are executed properly, or none. There must be no state in a database where a transaction is left partially completed. States should be defined either before the execution of the transaction or after the execution/abortion/failure of the transaction.
- **Consistency**. The database must remain in a consistent state after any transaction. No transaction should have any adverse effect on the data residing in the database. If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.
- **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.

Operation on Transactions:
Transactions access data using two operations:

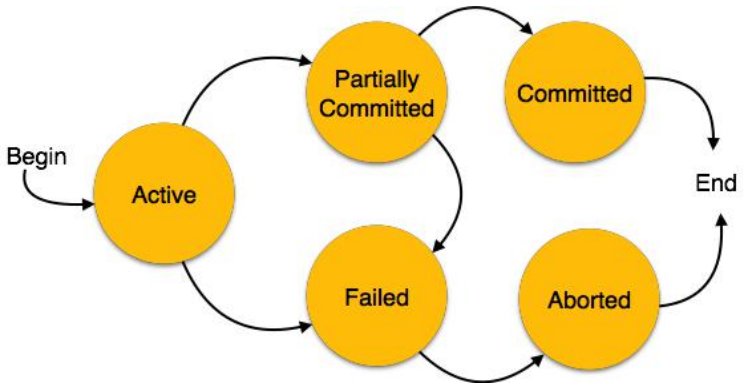
- **read(X),** which transfers the data item X from the database to a local buffer belonging to the transaction that executed the read operation.
- **write(X),** which transfers the data item X from the the local buffer of the transaction that executed the write back to the database.

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

```
Ti:   read(A);
      A := A − 50;
      write(A);
      read(B);
      B := B + 50;
      write(B).
```

Transaction State: 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.

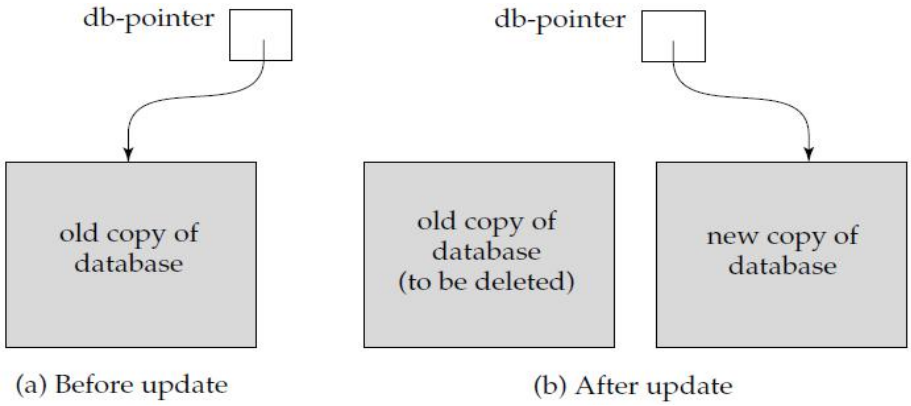


1. A transaction starts in the active state.
2. When it finishes its final statement, it enters the partially committed state. At this point, the transaction has completed its execution, but it is still possible that it may have to be aborted, since the actual output may still be temporarily residing in main memory, and thus a hardware failure may preclude its successful completion.
3. 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).
4. Failed 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.
5. If a transaction executes all its operations successfully, it is said to be committed. All its effects are now permanently established on the database system.

Implementation of Atomicity and Durability:

Shadow Copy: Shadow copy is a simple, but extremely inefficient scheme to maintain atomicity and durability of transaction. 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.



(a) Before update (b) After update

If the transaction completes, it is committed as follows.

1. The operating system is asked to make sure that all pages of the new copy of the database have been written out to disk.
2. 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;
3. The new copy then becomes the current copy of the database. The old copy of the database is then deleted.

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

Concurrent Executions

Transaction-processing systems usually allow multiple transactions to run concurrently. Allowing multiple transactions to update data concurrently causes several complications with consistency. Ensuring consistency in spite of concurrent execution of transactions requires extra work; it is far easier to insist that transactions run serially—that is, one at a time, each starting only after the previous one has completed.

There are two good reasons for allowing concurrency:

- **Improved throughput and resource utilization**: Concurrent transaction increases the throughput of the system—that is, the number of transactions executed in a given amount of time.
- **Reduced waiting time**: Concurrent transaction reduces the average response time: the average time for a transaction to be completed after it has been submitted.

Schedules: The execution sequences that describe chronological order in which instructions are executed in the system are called schedules.

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

T1: read(A);
A := A − 50;
write(A);
read(B);
B := B + 50;
	T2: read(A);
	temp := A * 0.1;
	A := A − temp;
	write(A);
	read(B);

write(B).	B := B + temp; write(B)

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.

Serial Schedule: Serial schedule consists of a sequence of instructions from various transactions, where the instructions belonging to one single transaction appear together in that schedule.
Now let first execution sequence T1 followed by T2 as schedule 1, and to the second execution sequence T2 followed by T1 as schedule 2.

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

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

Serial schedule must be in following order:
Serial schedule 1: r1(A), w1(A), r1(B), w1(B), r2(A), w2(A), r2(B), w2(B)
Serial schedule 2: r2(A), w2(A), r2(B), w2(B), r1(A), w1(A), r1(B), w1(B),

Concurrent Schedule: Concurrent schedule also known as non-serial schedule. A non-serial schedule is a schedule where the operations of a group of concurrent transactions are interleaved. Eg. Schedule-3

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

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

UNIT-4

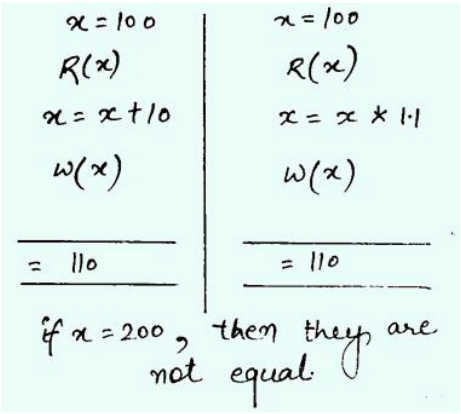
Serializability: Serializability is the classical concurrency scheme. It ensures that a schedule for executing concurrent transactions is equivalent to one that executes the transactions serially in some order. It assumes that all accesses to the database are done using read and write operations.

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

To ensure serializability, we consider only two operations: read and write. Between a read(Q) 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.

Equivalence Schedules
An equivalence schedule can be of the following types −

1. **Result Equivalence**
   If two schedules produce the same result after execution, they are said to be result equivalent. They may yield the same result for some value and different results for another set of values. That's why this equivalence is not generally considered significant.



Schedule 1			Schedule 2	
T1	T2		T1	T2
read(A) write (A) read(B) write(B)				read(A) write (A) read(B) write(B)
	read(A) write (A) read(B) write(B)		read(A) write (A) read(B) write(B)	

2. **View Equivalence**
Two schedules would be view equivalence if the transactions in both the schedules perform similar actions in a similar manner.
For example −
   - If T reads the initial data in S1, then it also reads the initial data in S2.
   - If T reads the value written by J in S1, then it also reads the value written by J in S2.
   - If T performs the final write on the data value in S1, then it also performs the final write on the data value in S2.

3. **Conflict Equivalence**
Two operations in a schedule would be conflicting if they have the following properties −
   - They belong to different transactions.
   - They access the same data item.
   - At least one of them is "write" operation.

Two schedules S1 and S2 having multiple transactions with conflicting operations are said to be conflict equivalent if and only if −
   - Both the schedules contain the same set of Transactions.
   - The order of conflicting pairs of operation is maintained in both the schedules.

UNIT-4

Conflict Serializability: Let us consider a schedule S in which there are two consecutive instructions Ii and Ij, of transactions Ti and Tj , respectively ($i \neq j$).

- 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.
- 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 considered:

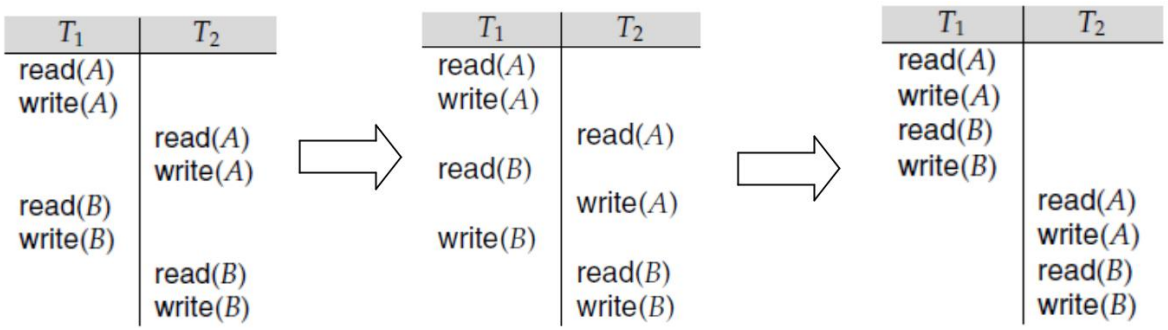
1. Ii = read(Q), Ij = read(Q). The order of Ii and Ij does not matter, since the same value of Q is read by Ti and Tj , regardless of the order.

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 1 where both Ii and Ij are read instructions does the relative order of their execution not matter.



Schedule1: showing only the read and write instructions. Schedule2

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.
For the above schedule1- We continue to swap non-conflicting instructions:
1. Swap the read(B) instruction of T1 with the read(A) instruction of T2.
2. Swap the write(B) instruction of T1 with the write(A) instruction of T2.
3. Swap the write(B) instruction of T1 with the read(A) instruction of T2.

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 final result of these swaps is a serial schedule. Thus, 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.

Recoverability:

1. Recoverable Schedules
A recoverable schedule is one where, for each pair of transactions Ti and Tj such that Tj reads a data item previously written by Ti, the commit operation of Ti appears before the commit operation of Tj

Let the schedule 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, T9 commits 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.

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

2. Cascadeless Schedules:
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.
A cascadeless schedule is one where, for each pair of transactions Ti and Tj such that Tj reads a data item previously written by Ti, the commit operation of Ti appears before the read operation of Tj . It is easy to verify that every cascadeless schedule is also recoverable. .

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

Consider the partial schedule Transaction T10 writes a value of A that is read by transaction T11. Transaction T11 writes a value of A that is read by transaction T12. Suppose that, 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 phenomenon, in which a single transaction failure leads to a series of transaction rollbacks, is called cascading rollback.

Recovery: Recovery scheme is an integral part of a database system that can restore the database to the consistent state that existed before the failure. The recovery scheme must also provide high availability; that is, it must minimize the time for which the database is not usable after a crash.

Failure Classification:
There are various types of failure that may occur in a system, each of which needs to be deal with in a different manner. The simplest type of failure is one that does not result in the loss of information in the system. The failures that are more difficult to deal with are those that result in loss of information. We shall consider only the following types of failure:

1. **Transaction failure**. There are two types of errors that may cause a transaction to fail:
    a. **Logical error**. The transaction can no longer continue with its normal execution because of some internal condition, such as bad input, data not found, overflow, or resource limit exceeded.
    b. **System error**. The system has entered an undesirable state (for example, deadlock), as a result of which a transaction cannot continue with its normal execution. The transaction, however, can be re-executed at a later time.
2. **System crash**. There is a hardware malfunction, or a bug in the database software or the operating system, that causes the loss of the content of volatile storage, and brings transaction processing to a halt. The content of nonvolatile storage remains intact, and is not corrupted. The assumption that hardware errors and bugs in the software bring the system to a halt, but do not corrupt the nonvolatile storage contents, is known as the fail-stop assumption. Well-designed systems have numerous internal checks, at the hardware and the software level that brings the system to a halt when there is an error. Hence, the fail-stop assumption is a reasonable one.
3. **Disk failure**. A disk block loses its content as a result of either a head crash or failure during a data transfer operation. Copies of the data on other disks, or archival backups on tertiary media, such as tapes, are used to recover from the failure.

Storage Structure

We have already described the storage system. In brief, the storage structure can be divided into two categories –

- **Volatile storage** − As the name suggests, a volatile storage cannot survive system crashes. Volatile storage devices are placed very close to the CPU; normally they are embedded onto the chipset itself. For example, main memory and cache memory are examples of volatile storage. They are fast but can store only a small amount of information.
- **Non-volatile storage** − these memories are made to survive system crashes. They are huge in data storage capacity, but slower in accessibility. Examples may include hard-disks, magnetic tapes, flash memory, and non-volatile (battery backed up) RAM.

Recovery and Atomicity

When a system crashes, it may have several transactions being executed and various files opened for them to modify the data items. Transactions are made of various operations, which are atomic in nature. But according to ACID properties of DBMS, atomicity of transactions as a whole must be maintained, that is, either all the operations are executed or none.

When a DBMS recovers from a crash, it should maintain the following −

- It should check the states of all the transactions, which were being executed.
- A transaction may be in the middle of some operation; the DBMS must ensure the atomicity of the transaction in this case.
- It should check whether the transaction can be completed now or it needs to be rolled back.
- No transactions would be allowed to leave the DBMS in an inconsistent state.

There are two types of techniques, which can help a DBMS in recovering as well as maintaining the atomicity of a transaction −

- Maintaining the logs of each transaction, and writing them onto some stable storage before actually modifying the database.
- Maintaining shadow paging, where the changes are done on a volatile memory, and later, the actual database is updated.

UNIT-4

Log-based Recovery

Log is a sequence of records, which maintains the records of actions performed by a transaction. It is important that the logs are written prior to the actual modification and stored on a stable storage media, which is failsafe. 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 . Xj had value V1 before the write, and will have value V2 after 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. The database can be modified using two approaches

- **Deferred database modification** − All logs are written on to the stable storage and the database is updated when a transaction commits.
- **Immediate database modification** − Each log follows an actual database modification. That is, the database is modified immediately after every operation.

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.

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>.

Checkpoints

There are two major difficulties with this Log Based Recovery and redo /undo operation.

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.

Thus keeping and maintaining logs in real time and in real environment may fill out all the memory space available in the system. As time passes, the log file may grow too big to be handled at all.

To reduce these types of overhead, we introduce checkpoints. Checkpoint is a mechanism where all the previous logs are removed from the system and stored permanently in a storage disk. Checkpoint declares a point before which the DBMS was in consistent state, and all the transactions were committed. 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>.

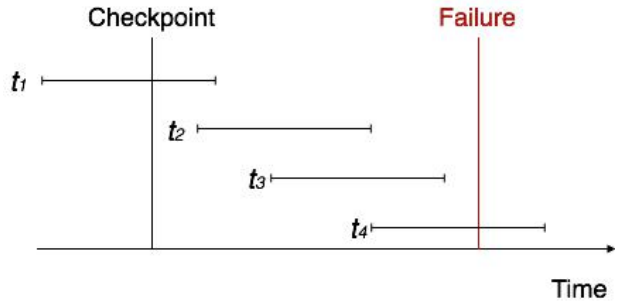
Recovery

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 <Tk commit> record in the log, execute undo(Tk).
- For all transactions Tk in T such that the record <Tk commit> appears in the log, execute redo(Tk).

When a system with concurrent transactions crashes and recovers, it behaves in the following manner −

- The recovery system reads the logs backwards from the end to the last checkpoint.
- It maintains two lists, an undo-list and a redo-list.
- If the recovery system sees a log with <T_n, Start> and <T_n, Commit> or just <T_n, Commit>, it puts the transaction in the redo-list.
- If the recovery system sees a log with <T_n, Start> but no commit or abort log found, it puts the transaction in undo-list.



All the transactions in the undo-list are then undone and their logs are removed. All the transactions in the redo-list and their previous logs are removed and then redone before saving their logs.

Deadlock

A system is in a deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set. More precisely, there exists a set of waiting transactions {T0, T1, . . ., Tn} such that T0 is waiting for abdata item that T1 holds, and T1 is waiting for a data item that T2 holds, and . . ., and Tn−1 is waiting for a data item that Tn holds, and Tn is waiting for a data item that T0 holds. None of the transactions can make progress in such a situation.

Deadlock Handling

There are two principal methods for dealing with the deadlock problem.

1. Deadlock prevention protocol
2. Deadlock detection and deadlock recovery

1. Deadlock Prevention:

There are Two different deadlock prevention schemes using timestamps have been proposed

(a). **The wait–die scheme** is a non preemptive technique. When transaction Ti requests a data item currently held by Tj , Ti is allowed to wait only if it has a timestamp smaller than that of Tj (that is, Ti is older than Tj). Otherwise, Ti is rolled back (dies).
For example, suppose that transactions T1, T2, and T3 have timestamps 5, 10, and 15, respectively. If T1 requests a data item held by T3, then T1 will wait. If T3 requests a data item held by T2, then T3 will be rolled back.

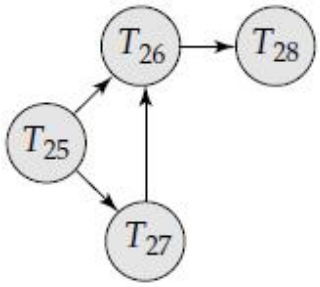
(b). **The wound–wait scheme** is a preemptive technique. It is a counterpart to the wait–die scheme. When transaction Ti requests a data item currently held by Tj , Ti is allowed to wait only if it has a timestamp larger than that of Tj (that is, Ti is younger than Tj). Otherwise, Tj is rolled back (Tj is wounded by Ti).
For example, with transactions T1, T2, and T3, if T1 requests a data item held by T1, then the data item will be preempted from T2, and T2 will be rolled back. If T3 requests a data item held by T2, then T3 will wait.

2. Deadlock Detection and Recovery:

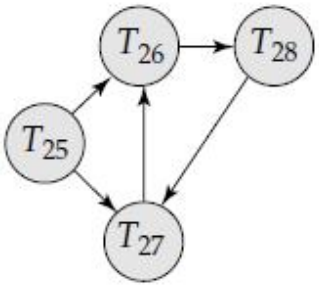
If a system does not employ some protocol that ensures deadlock freedom, then a detection and recovery scheme must be used. An algorithm that examines the state of the system is invoked periodically to determine whether a deadlock has occurred.

Deadlock Detection:

A deadlock exists in the system if and only if the wait-for graph contains a cycle. Each transaction involved in the cycle is said to be deadlocked. To detect deadlocks, the system needs to maintain the wait-for graph, and periodically to invoke an algorithm that searches for a cycle in the graph.



Wait-for graph without a cycle.



Wait-for graph with a cycle.

Wait-for-Graph

Deadlocks can be described precisely in terms of a directed graph called a wait-for graph. This graph consists of a pair G = (V, E), where V is a set of vertices and E is a set of edges. The set of vertices consists of all the transactions in the system. Each element in the set E of edges is an ordered pair Ti → Tj. If Ti → Tj is in E, then there is a directed edge from transaction Ti to Tj , implying that transaction Ti is waiting for transaction Tj to release a data item that it needs.

When transaction Ti requests a data item currently being held by transaction Tj , then the edge Ti → Tj is inserted in the wait-for graph. This edge is removed only when transaction Tj is no longer holding a data item needed by transaction Ti.

Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, the system must recover from the deadlock. The most common solution is to roll back one or more transactions to break the deadlock. Three actions need to be taken:

UNIT-4

1. **Selection of a victim.** Given a set of deadlocked transactions, we must determine which transaction (or transactions) to roll back to break the deadlock.We should roll back those transactions that will incur the minimum cost. Many factors may determine the cost of a rollback, including
   a. How long the transaction has computed, and how much longer the transaction will compute before it completes its designated task.
   b. How many data items the transaction has used.
   c. How many more data items the transaction needs for it to complete.
   d. How many transactions will be involved in the rollback.

2. **Rollback**: Once we have decided that a particular transaction must be rolled back, we must determine how far this transaction should be rolled back.
   (a). Total rollback: Abort the transaction and then restart it. However, it is more effective to roll back the transaction only as far as necessary to break the deadlock.
   (b). Partial rollback requires the system to maintain additional information about the state of all the running transactions. Specifically, the sequence of lock requests/grants and updates performed by the transaction needs to be recorded.

3. **Starvation**. In a system where the selection of victims is based primarily on cost factors, it may happen that the same transaction is always picked as a victim. As a result, this transaction never completes its designated task, thus there is starvation. We must ensure that transaction can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.