# Introduction to Transactions:

**UNIT-V TRANSACTION MANAGEMENT**

**Def**: A transaction is an execution of a user program and is seen by the DBMS as a series or list of actions. The actions that can be executed by a transaction includes the reading and writing of database. **Transaction Operations:** Access to the database is accomplished in a transaction by the following two operations.

1. **Read(X):** Performs the reading operation of data item X from the database.
2. **Write(X):** Perform the writing operation of data item X to the database.

**Example of Transaction:** Let T1 be a transaction that transfers $50 from account A to account B. This transaction can be illustrated as follows.

# T1: read(A);

**A:= a – 50;**

# Write(A);

**Read(B); B:=B + 50;**

# Write(B);

**Transaction Concept**: The concept of transaction is the foundation for concurrent execution of transactions in a DBMS and recovery from system failure in a DBMS. In this case, high level language supports DBMS with respect to concurrency control and recovery of data. It is convenient to execute the user program or transactions (database object) with the help of read and write operations.

1. To read a database object, the read operation first brought the database object into main memory from disk.
2. To write a database object, the write operation first modifies the copy of object in main memory and then writes on disk.

Generally, database objects are the units in which programs reads or writes information. The units could be pages, records and so on but these executes depends on the DBMS.

**PROPERTIES OF TRANSACTION (ACID):** There are four important properties of transaction that a DBMS must ensure to maintain data in concurrent access of database and recovery from system failure in DBMS. They are acronym of ACID.

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

1. **Atomicity:** Atomicity ensures that the transaction either is executed completely or not at all. It means, users should not have to worry about the effect of incomplete transactions when a system crash occurs.

Transactions can be incomplete for three kinds of reasons. They are

* 1. First, a transaction can be aborted or terminated unsuccessfully by the DBMS because some changes arise during execution. If a transaction is aborted by the DBMS for some internal reason, it is automatically restarted and executed as a new transaction.
  2. Second, the system may crash because the power supply is interrupted, while one or more transactions are in progress.
  3. Third, a transaction may encounter an unexpected situation such as unable to access disk due to virus, etc.

1. **Consistency:** Consistency means, data in the database must always in a consistent state i.e. available. Execution of transaction in isolation preserves the consistency of the database.

This property is the authority of the application programmer. It means, if the programmer wants some data to be consistent then he gives the consistency permission to that data.

For example, consider a transaction that involves transfer of amount. If amount is debited from account A and credited to account B, then the two accounts A & B must be consistent. If one is not available then transaction is aborted. So, this is the responsibility of application program to ensure consistency. Because, before start the transaction and after the transaction, the account A & B must be consistent.

1. **Isolation:** Isolation property ensures that each transaction is unaware (not known) of other transactions executing concurrently in the system.

For example, suppose multiple transactions are executing concurrently in the system such as T1 and T2. During the process, the transaction of T1 is unaware of T2 i.e. whether T1 has started or finished or not but T2 is unaware of T1. It means, T2 doesn’t know the details of T1 during the transaction.

1. **Durability:** This property ensures that data remains in a consistent (available) state even after the failure. This is ensured by loading the modified data into disk. It means, the durability property guarantees that, once a transaction completes successfully, all the updates that are carried out on the database even if there is a system failure after the transaction completes execution.

For example, we can assume that a failure of the computer system may result in loss of data in main memory (buffer) but data written to disk is never lost.

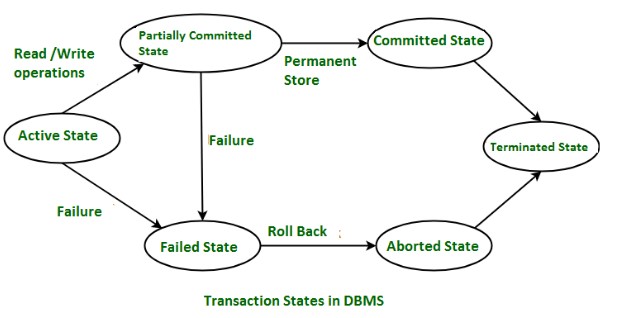
 In this case, durability can ensure the transactions following two reasons,

* 1. First the updates carried out by the transaction have been written to disk before the transaction completes.
  2. Second, 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.

Thus, ensuring durability is the responsibility of a component of the database system called the recovery-management component. Moreover, the transaction-management component and the recovery-management component are closely related.

**TRANSACTION STATE**: A transaction is an execution of a user program and is seen by the DBMS as a series (list) of actions. These can be established by a simple transaction model named a s transaction states.

1. **Active State:** This is the initial state of a transaction. The transaction stays in this state while it is execution.
2. **Partially committed state:** This transaction state occurs after the final (last) statement of the transaction has been executed.
3. **Failed State:** This transaction state occurs after the discovery that normal execution can no longer proceed.
4. **Aborted State:** This transaction state occurs after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction.
5. **Committed State:** This transaction state occurs after the successful completion of the transaction. The transaction states diagram corresponding to a transaction is shown in fig.



A transaction starts in the active state. When it finishes its final (last) statement, it enters the partially committed state. At this point, the transaction has completed its execution in main memory and it is possibility to **abort** due to hardware failure. In this case, the transaction which is temporarily resided in main memory will be lost. So the transaction is restarted.

A transaction may also enters the failed state from the active state or from the partially committed state due to the hardware failure or logical errors, the transaction can be restarted. At this state, the system has two operations. Such as

* 1. **Restart the Transaction:** It can restart the transaction, but only if the transaction was aborted as a result of some hardware failure or software error. A restarted transaction is considered to be a new transaction.
  2. **Kill the Transaction:** It can kill the transaction because of some internal logical error that can be corrected only by rewriting the application program, or because the input was bad.

**CONCURRENT EXECUTION:** In transaction processing system, multiple transactions can be executed concurrently. Concurrent execution of transaction in database is similar to the multiprogramming approach performed in operating system. However, such execution results in inconsistency of data. To ensure data consistency during the concurrent execution of several transactions, an extra effort is required. This problem can be solved by employing different mechanisms are referred as concurrency control schemes. They are

1. Increase the efficiency/ throughput of system. 2. Decrease the response time and average time.

1. **Increase the efficiency/ throughput of system:** A transaction may contain either I/O activity or CPU activity. Both of these activities can be performed parallel. This parallelism enables the system to execute multiple transactions simultaneously. Such mechanism increases the efficiency when more number of transactions is executed at a time.
2. **Decreases the Response Time and Average Time:** Transactions can either be short or long. If both these transactions are executed concurrently, then short transaction need not wait for the long transaction to complete its execution.

**Types of Schedules for transaction execution concurrently:** They are different types of schedules for transaction execution. They are

* 1. **Serial and Non-Serial Schedule**: A schedule is a list of actions (such as reading, writing, aborting or committing) from set of transactions.

|  |  |  |  |
| --- | --- | --- | --- |
|  | T1 | T2 |  |
| read(x) write(x) read(y) write(y) | read(x) write(x) read(x) write(x) |
| T1 | | T2 | |
| Read(sam) Sam:=sam – 200 | |  | |
|  | | Read(sam)  Sam:=sam –sam \* 20/100 Write(sam)  Read(joan) | |
| Write (sam) Read(joan) Joan:=joan + 200  Write(joan) | |  | |
|  | | Joan:=joan+sam\*20/100 Write (joan) | |

 **Serial schedule** is a schedule for transactions that are executed one after another sequentially. All the transactions are appeared in serial schedule. The number of serial schedules generated for a given schedule depends on the number of transactions.

Consider the example, T1 schedule has serial transaction and T2 has serial transactions.

 **Non-Serial schedule**, the multiple transactions are executed not in a serial schedule. In concurrent execution, the operating system initially executes

few transactions of first transaction at T1 and the CPU switches to executes the instruction of second transaction at T2. Later it switches back to first transaction and executes the remaining instructions and so on.

Thus, In concurrent execution, transactions may be interleaved. Due to this, there is a possibility that more than one execution sequence may exist.

Since, the execution of transaction in Non-serial schedule is incorrect state to find the sum of both accounts.

|  |  |
| --- | --- |
| T1 | T2 |
| read(x) |  |
| write(x) |  |
|  | read(y) |
|  | write(y) |
| read(c) |  |
| write(c) |  |

* 1. **Anomalies due to Interleaved Execution:** The schedule, Involving on two transactions is shown in fig. that represents an interleaved execution of the two transactions.

First, If one transaction is waiting in one schedule to be read from Disk, the CPU can process another transaction. This is because

I/O activity can be done in parallel with CPU activity in a computer. In this case, I/O activity and CPU activity reduces its time and complete the transaction.

Second, interleaved execution of a short transaction with a long transaction usually allows the short transaction to complete quickly. In this case, three anomalies associated with interleaved execution on the same data object. They are

1. **Write-Read(WR) Conflict**: Reading Uncommitted data.
2. **Read-Write(RW) Conflict:** Unrepeatable reads.
3. **Write-Write(WW) Conflict:** Overwriting Uncommitted Data.

**SERIALIZABILITY:** Serializability is a widely accepted standard that ensures the consistency of schedule. A schedule is consistent if and only if it is serializable. A schedule is said to be serializable if the interleaved transactions produces the result, which is equivalent to the result produced by executing individual transactions separately.

|  |  |
| --- | --- |
| T1 | T2 |
| read(x) |  |
| write(x) |  |
| read(y) |  |
| write(y) |  |
|  | read(x) |
|  | write(x) |
|  | read(y) |
|  | write(y) |

|  |  |
| --- | --- |
| T1 | T2 |
| read(x) write(x) |  |
|  | read(y)  write(y) |
| read(y) write(y) |  |
|  | read(x) write(x) |

fig1 serial schedule fig2 interleaved schedule

The above two schedules produce the same result, these schedules are said to be serializable. The transaction may be interleaved in any order and DBMS doesn’t provide any guarantee about the order in which they are executed. The two different types of serializability are

1. Conflict Serializability 2) View Serializability

1. **Conflict Serializability**: The following schedules represents disagree of serializability.

# Write-Read (WR) Conflict: Reading Uncommitted Data (WR Conflicts) (or) Dirty Read:

The first source of anomaly is that a transaction T2 will start the execution if T1 transaction execution is committing (or) not (or) aborted.

|  |  |
| --- | --- |
| T1 | T2 |
| read(x) write(x)  Abort/commit | read(x) write(x) Abort/commit |

Consider two transactions T1 and T2. The T1 adding amount to the account of x and T2 subtract the amount of x. It means,

 First, T1 transaction is completed in RAM but not committed to write on disk.

 Second, the control is switched to start the T2 transaction execution immediately after its process is over.

and it is aborted/committed

 Thirdly, Now T1 transaction may be committed (or) aborted.

Thus, this causes dirty read because, T2 execution is started before commit/abort of T1 execution.

1. **Read-Write(RW) Conflict**: **Unrepeatable Reads Conflict):** The second source of anomaly is that a transaction T2 is committed before commit of T1 transaction. In this case, the read operation can be read same data but finally appears only one data. It means,

|  |  |  |
| --- | --- | --- |
| T1 | T2 |  |
| read(x) write(x)  abort | read(x) write(x) commit | |

 First, T1 transaction is completed but not committed.

Second, T2 schedule transaction over and it is committed.

Third, T1 schedule is aborted. It means, it start its process from beginning.

Thus, first T2 transaction is written on disk and after that T1 schedule transaction is aborted. It means, T1 restart its transaction but read current data not previous data. This causes un-repeatable read conflict and dirty write.

1. **Write-Write(WW) Conflict**: **Overwriting Uncommitted Data (or) Blind Writes**: The source of anomalies is that a transaction T1 could overwrite the value of an

|  |  |
| --- | --- |
| T1 | T2 |
| read(x) write(x)  commit | read(x) write(x) commit |

object which is already modified by a transaction T2 while T1 is still in progress. This is shown in fig.

1. **VIEW SERIALIZABILITY**: Two schedules S1 and S2 consisting of same set of transactions are said to be view equivalent, if the following conditions are satisfied.
   1. If a transaction T1 in Schedule S1 performs the read operation on the initial value of data item x, then the same transaction in schedule S2 must also perform the read operation on the initial value of x.
   2. If a transaction T1 in schedule S1 reads the value x and that was written by transaction T2 in S1, then it must read the value x in S2 written by transaction t2.
   3. If a transaction T1 in schedule S1 performs the final write operation on data item x, then the same transaction in schedule S2 must also perform the final write operation on x.

Consider the following schedules that are view equivalent.

Schedule S1 Schedule S2

|  |  |
| --- | --- |
| T1 | T2 |
| read(x) x:=x-10  write(x) |  |
|  | read(x)  x:=x\*20 write(x) |
| read(y) y:=y+10  write(y) |  |
|  | read(y) y:=y/20 write(y) |

|  |  |
| --- | --- |
| T1 | T2 |
| read(x) |  |
| x:=x-10 |  |
| write(x) |  |
| read(y) |  |
| y:=y+10 |  |
| write(y) |  |
|  | read(x) |
|  | x:=x\*20 |
|  | write(x) |
|  | read(y) |
|  | y:=y/20 |
|  | write(y) |

Thus, every conflict serializable schedule is view serializable but every view serializable schedule is not conflict serializable.

# CONCURRENCY CONTROL:

When multiple transactions are trying to access the same sharable resource, there could arise many problems if the access control is not done properly. There are some important mechanisms to which access control can be maintained. Earlier we talked about theoretical concepts like serializability, but the practical concept of this can be implemented by using **Locks** and **Timestamps**. Here we shall discuss some protocols where Locks and Timestamps can be used to provide an environment in which concurrent transactions can preserve their Consistency and Isolation properties.

LOCK BASED PROTOCOLS: To ensure serializability it is required that data item should be accessed in mutual exclusive manner. If one transaction is accessing a data item, no other transaction can modify that data item. to implement this requirement locks are used a transaction is allowed to access a data item only if it is currently holding a lock on data item.

There are two modes in which a data item may be locked.

**Shared mode lock**: if a transaction Ti has obtained a shared mode lock on item Q, then Ti can read but cannot write Q. It is denoted by S.

**Exclusive mode lock**: if a transaction Ti has obtained an exclusive mode lock on item Q, then Ti can read and also write Q. It is denoted by X.

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.

Given a set of lock modes, we can define a **compatibility function** on them as follows Let A and B represent arbitary lock modes. Suppose that a - transaction Ti requests a lock of mode A on item Q on which transaction Ti (Ti #Ti ) currently hold a lock of mode B. if transaction T1 can be granted a lock of 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 is represented by a matrix. The matrix is shown in.

|  |  |  |
| --- | --- | --- |
|  | S | X |
| S | true | false |
| X | false | false |

**Example:** Transaction 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);

Transaction T2: lock-S(A);

Read(A);

Unlock(A);

Lock-S(B);

Read(B); Unlock(B); Display(A+B);

# The Two Phase Locking Protocol:

This protocol requires that each transaction issue lock and unlock requests in two phases.

1. **Growing phase**: In this phase, a transaction may obtain locks, but may not release any lock.
2. **Shrinking phase**: In this phase, a transaction may release locks, but may not obtain any new lock.

Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters in the shrinking phase, and it cannot issue more lock requests.

Transaction T1 is two phase. T1 : lock-X(B);

read (B);

B := B - 50;

Write (B);

Lock-X(A);

read (A);

A: = A + 5:);

write (A); unlock (B); unlock(A);

**Advantage :**The two-phase locking protocol ensures conflict serializability. Consider any transaction, the point in the schedule where the transaction has obtained its final lock is called the lock point of the transaction. Now, transaction can be ordered according to their lock points. This ordering is a serializability ordering for the transactions.

**Disadvantages:** i) Cascading rollbacks may occur under two-phase locking. Consider schedule 2 shown in Fig

**Strict two-phase locking protocol:** This protocol requires that locking should be two phase, and all exclusive-mode locks taken by a transaction should be held until the transaction. This requirement prevents any transaction from reading the data written by any uncommitted transaction under exclusive mode until the transaction commits.

**The rigorous two phase locking protocol**: This protocol requires that all locks be held until the transaction commits.

# TIMESTAMP BASED PROTOCOLS:

Time stamp based protocol ensures serializabililty. It selects an ordering among transactions in advance using time stamps.

**Timestamps**: With each transaction in the system, a unique fixed timestamp is associated. It is 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).

Two methods are used for implementing timestamp:

1. Use the value of the system clock as the timestamp, that is, a timestamp is equal to the value of the clock when the transaction system.
2. Use a logical counter, that is a transactions timestamp is equal to logical counter, when transaction enters the system. After assign a new timestamp, value of timestamp is increased.

The timestamps of the transactions determine the serializability order. Thus if TS(Ti )>TS(Tj) , then the system must ensure that in produced schedule transaction Ti appears before transaction Tj

To implement this scheme, two timestamps are associated with each data item Q.

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

These timestamps are updated whenever a new read(Q) or write(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 :

# Suppose that transaction Ti issues read(Q).

1. If TS(Ti) < W-timestamp(Q), then Ti needs a value of Q that was already overwritten. Hence, read operation is rejected, and Ti is rolled back.
2. If TS(Ti)>=W-timestamp(Q), then the read operation is executed, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(Ti).

# Suppose that transaction Ti issues write(Q).

1. If TS(Ti) < R-timestamp(Q), then the value of Q that Ti is producing was needed previously, and the system assumed that the value would never be produced. Hence, the system rejects 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, the system rejects this write operation and rolls back Ti.
3. Otherwise, the system executes the write operation and sets W-timestamp(Q) to TS(Ti).

If a transaction Ti is rolled by the concurrency control scheme, the system assigns it a new timestamp and restarts it.

Example: consider two transactions T1 and T2. T1 display the sum of account A and B and transaction T2 transfer $50 amount from account B to account A and display the sum of both.

T1: read(A); Shows a concurrent schedule for these two transactions.

Read(B); Display(A+B);

|  |  |
| --- | --- |
| T1 | T2 |
| read(B) |  |
|  | read(B) |
|  | B:=B-50 |
|  | Write(B) |
| read(A) |  |
|  | read(A) |
| display(A+B) |  |
|  | A:=A+50 |
|  | Write(A) |
|  | Display(A+B) |

T2: read(B);

B:=B-50;

Write(B);

Read(A); A:=A+50;

Write(A); Display(A+B);

**Advantages:** the timestamp ordering protocol ensure conflict serialzability this is because conflicting operations are processed in timestamp order.

**Disadvantages:** There is a possibility of starvation of long transaction if a sequence of conflicting short transaction causes repeated restarting of the long transactions.

**Thomas write Rule**

Thomas write rule is a modified version of timestamp ordering protocol. consider schedule given following

|  |  |  |
| --- | --- | --- |
| T1 | T2 |  |
| read(x)  write(x) | write(x) |

here T1 stars before T2, therefore TS(T1)<TS(T2) the read(x) operation of T1 succeeds, similarly the write(x) operation of T2. when T1 attempts its write(x) operations it is rejected by the system and T2 is rolled back; as (Ts(T1)<W-timestamp(x)) since W-timestamp(x)=TS(T2).

in this case, T2 has already written X and the value of X that T 1 is attempting to write is one that will never need to be read. thus the rollback of T1 is required by timestamp ordering protocol but it is unnecessary.

Thomas write rule modifies the timestamp ordering protocol.

**Thomas write rule is:**

Suppose that transaction Ti issues write(Q):

1. If TS(Ti) < R-timestamp(Q), then the value of Q that Ti is producing was needed previously, and the system assumed that the value would never be produced. Hence, the system rejects 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, the system rejects this write operation and rolls back Ti.
3. Otherwise, the system executes the write operation and sets W-timestamp(Q) to TS(Ti).

**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. In other words, there exists a set of waiting transactions

{To, T1, ,Tn} such that To is waiting for a data item that T1 holds, and Ti 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 To holds. In such situation, none of transaction can make progress."

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

1. **Deadlock prevention** : This approach ensures that system will never enter in deadlock state.
2. **Deadlock detection and recovery**: This approach tries to recover from deadlock if system enters in deadlock state.

# DEADLOCK PREVENTION

There are two approaches for deadlock prevention :

* 1. One approach ensures that no cyclic waits can occur by ordering the requests for locks, or requiring all locks to be acquired together. This approach requires that each transaction locks all data items before it begins execution. It is required that, either all data items should be locked in one step, or none should be locked.

Disadvantages of this approach are

1. It is hard to predict before the transaction begins, what data items need to be locked.
2. Data-item utilization may be very low, since many of the data items may be locked but unused for a long time.
   1. The second approach for deadlock prevention is to use preemption and transaction rollbacks. In preemption when a transaction T2 requests a lock that transaction T1 holds, the lock granted to T1 may be preempted by rolling back T1, and granting of lock to T2. To control preemption, a unique timestamp is assigned to each transaction. The system uses timestamp to decide whether a transaction should wait or roll back.

Two different deadlock prevention schemes using timestamp are

# Wait die

The wait-die scheme is non preemption technique. In this, when transaction Ti. requests a data item held by Tj ,Ti is allowed to wait only if it has a timestamp smaller than Tj. (i.e. Ti is older than Tj). Otherwise, Ti is rolled back (dies).

For example, consider three transactions T1 , T2 and T3 with timestamps 5, 10, and 15 respectively. If T1 requests a data item held by T2, then T1 will wait. If T3 requests data item held by T2, then T2 will be rolled back.

# Wound wait

The wound-wait is preemptive technique. In this, when transaction Ti requests data item held by Tj, Ti is allowed to wait, only if it has timestamp greater than Tj (i.e Ti is younger than Tj . Otherwise Tj is rolled back.

Returning to same example, if T1 requests a data item held by T2 , then the data item will be preempted by T2, and T2 will be rolled back. If T3; requests a data item held by T2 then T3 will wait.

# Timeout Based Schemes

This approach for deadlock handling is based on lock timeouts. In this approach, a transaction that has requested a lock waits for at most a specified amount of time. If the lock has not been granted within that time, the transaction is said to be time out, and it rolls back itself and restarts. Thus, if there was a deadlock one or more transactions involved in the deadlock will time out and roll back, allowing the others to proceed.

# Advantages

* + This scheme is easy to implement.
  + It works well if transactions are short, and if long, waits are likely to be due to deadlocks.

# Disadvantages

It is hard to decide how long a transaction should wait. Too long waits results in unnecessary delays once a deadlock has occurred, and too short waits results in transaction rollbacks even when there is no deadlock.

Starvation is also possible with this scheme.

# DEADLOCK DETECTION AND RECOVERY

This approach uses an algorithm that examines the state of the system periodically to determine whether a deadlock has occurred. If one has occurred, then the system attempts to recover from the deadlock

# Deadlock Detection

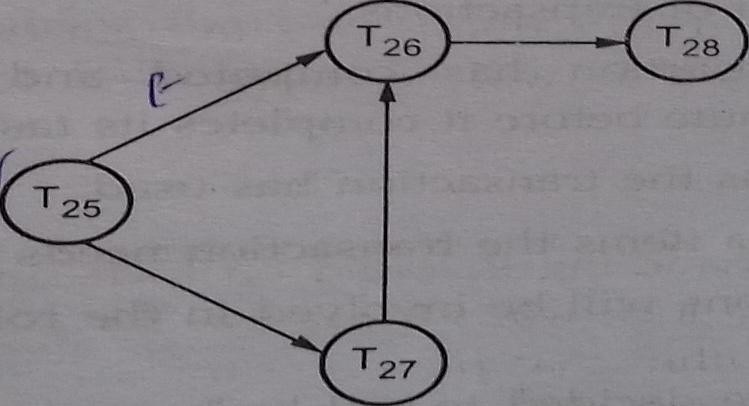
Deadlocks can be described in terms of directed graphs called a **wait-for graph**. This graph consists of a pair G =< V, E>, where

V - set of vertices consists of all transactions in the system E - set of edges

If Ti >Tj is in E, then there is a directed edge from transaction Ti to Tj. When transaction Ti requests

a data item currently held by transaction Tj then the edge TiTj is inserted in the wait for graph. 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 dead locked.

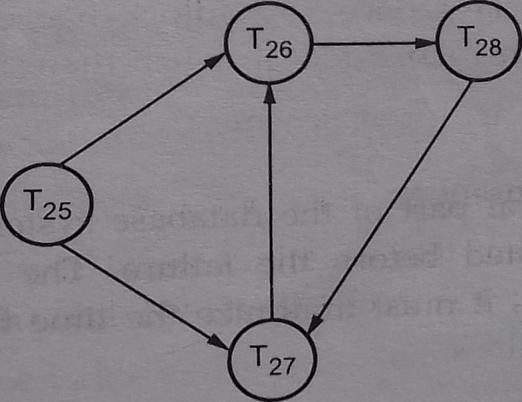
Example : Consider the wait for graph



The graph depicts following situation

* Transaction T25 is waiting for transactions T26 and T27.
* Transaction T27 is waiting for transaction T26.
* Transaction T26 is waiting for transaction T28.

This graph has no cycle; therefore the system is not in a deadlock state. Consider the graph



Above the graph contains a cycle' -

T26  T28T27T26

Therefore, the system is in deadlock state. Recovery from Deadlock

When a deadlock 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.

Three actions need to be taken:

1. **Selection of transactions**: Given a set of deadlocked transactions, we must determine which transactions should be rolled back to break the deadlock. The method used for this is : rollback those transactions that will incur minimum cost. Many factors determine the cost of transactions.
   * How long the transaction has computed and how much longer the transaction will compute before it completes its task.
   * How many data items the transaction has used.
   * How many more data items the transaction needs to complete its task.
   * How many transactions will be involved in the rollback.
2. **Rollback**: Once we have decided to roll back particular transaction, we must determine how far transaction should be rolled back. The solutions are

**Total rollback**: Abort the transaction and then restart it.

**Partial rollback**: It is more effective to roll back the transaction only as far as necessary to break the deadlock.

1. **Starvation**: It is possible that same transaction will be rolled back number of , times to break the deadlock. As a result, this transaction never completes its designated task. Thus, there is starvation. To avoid this, we must ensure that transactions can be picked as a victim only a small number of times.

# RECOVERY TECHNIQUES:

**Recovery System:** recovery system is an integral part of the database system. it stores the database to the consistent state that existed before the failure. The recovery system should 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.

# Transaction failure

There are two types of error that may cause transaction to fail.

**Logical error**: Logical error occurs because of some internal condition, such as bad input data not found, overflow or resource limit exceeded. When logic error occurs, transaction cannot continue with its normal execution.

**System error**: Example of system error is deadlock. When system error occurs, the system enters in an undesirable state, and as a result, transaction cannot continue with its normal execution.

# System crash:

There is a hard wave malfunction, or a bug in the database software or in 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.

is not corrupted.

# Disk failure:

A disk block loses its content as a result o either a head crash or failure during a data transfer operations. Copies of the data on other disks or archival backups on tertiary media, such as tapes are used to recover from the failure.

# Log-based Recovery:

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

There are several types of log records such as:-

* 1. **Update log record**: it describes a single database write. It has fallowing 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 the disk of the data item.

**Old values**: is the value of the data item prior to the write.

**New value**: is the value of the data item that it 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.

Various types of log records are represented as:

<Ti start >: Transaction Ti, has started.

<Ti, Xj,V1,V2>: Transaction Ti has performed a write on data item Xj. Xi had value 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, a log record for that write is created. Once a log 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.