UNIT III TRANSACTIONS

Transaction Concepts – ACID Properties – Schedules – Serializability – Concurrency Control – Need for Concurrency – Locking Protocols – Two Phase Locking – Deadlock – Transaction Recovery - Save Points – Isolation Levels – SQL Facilities for Concurrency and Recovery

TRANSACTION CONCEPTS

A **transaction** is a collection of operations that forms single logical unit of work.

Simple Transaction Example

1. Read your account balance
2. Deduct the amount from your balance
3. Write the remaining balance to your account
4. Read your friend's account balanace
5. Add the amount to his account balance
6. Write the new updated balance to his account

This whole set of operations can be called a transaction

Transaction processing system

- The system with large database and hundreds of concurrent users that are executing database transaction.
- Eg :reservation system , banking system etc

Concurrent access

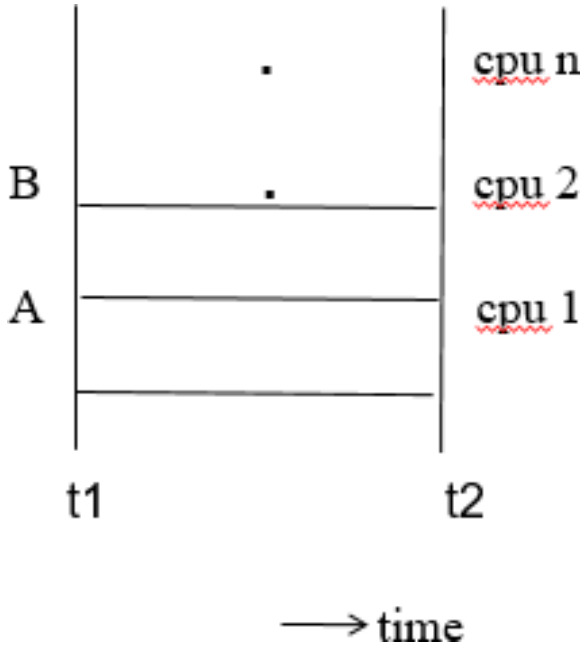
- mutiple user accessing a system at the same time

Single user-one user at a time can use a system

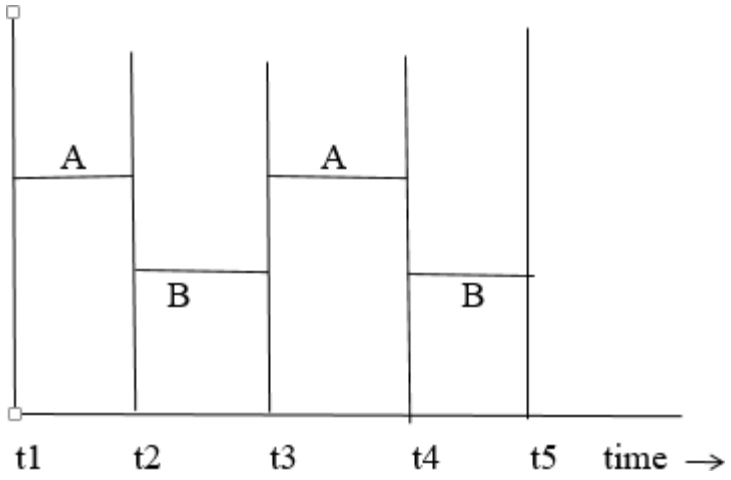
Multi user-many user use the system at a time.

It can be achieved by multiprogramming:

Parallel- multi-users access different resources at the same time.



Interleaved- Multiple users access a single resource based on time.



Transaction access data using two operations

- Read(x)

It transfer the data item **x** from the database to a local buffer belonging to the transaction that executed the read operation.

- Write(x)

It transfer the data item **x** from the local buffer of the transaction to the database i.e. it write back to the database.

ACID Properties

To ensure the integrity of data during a transaction, the database system maintains the following properties. These properties are widely known as ACID properties:

- **Atomicity** − This property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed 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.
- **Durability** − The database should be durable enough to hold all its latest updates even if the system fails or restarts. If a transaction updates a chunk of data in a database and commits, then the database will hold the modified data. If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system springs back into action.
- **Isolation** − In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.

E.g. transaction to transfer $50 from account A to account B:

1. **read**(*A*)
2. *A* := *A* – 50
3. **write**(*A*)
4. **read**(*B*)

5. $B := B + 50$
6. **write**(B)

Example of Fund Transfer

- **Atomicity requirement**
  - if the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state
  - the system should ensure that updates of a partially executed transaction are not reflected in the database
- **Durability requirement**
  - once the user has been notified that the transaction has completed, the updates to the database by the transaction must persist even if there are software or hardware failures.
- **Isolation requirement** — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database **T1** **T2**

1. **read**(A)
2. $A := A - 50$
3. **write**(A)

read(A), read(B), print(A+B)

4. **read**(B)
5. $B := B + 50$
6. **write**(B)

- Isolation can be ensured trivially by running transactions **serially**
  - that is, one after the other.

SCHEDULES

- Schedule – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.
- **Serial Schedule**

  It is a schedule in which transactions are aligned in such a way that one transaction is executed first. When the first transaction completes its cycle, then the next transaction is executed. Transactions are ordered one after the other. This type of schedule is called a serial schedule, as transactions are executed in a serial manner.

  Schedule 1
  - Let T_1 transfer 50 from A to B, and T_2 transfer 10% of the balance from A to B.
  - A serial schedule in which T_1 is followed by T_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 2

- A serial schedule where T_2 is followed by T_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 3

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1.

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 4

The following concurrent schedule does not preserve the value of ($A + B$).

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)

SERIALIZABILITY

When multiple transactions are being executed by the operating system in a multiprogramming environment, there are possibilities that instructions of one transactions are interleaved with some other transaction.

- 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.

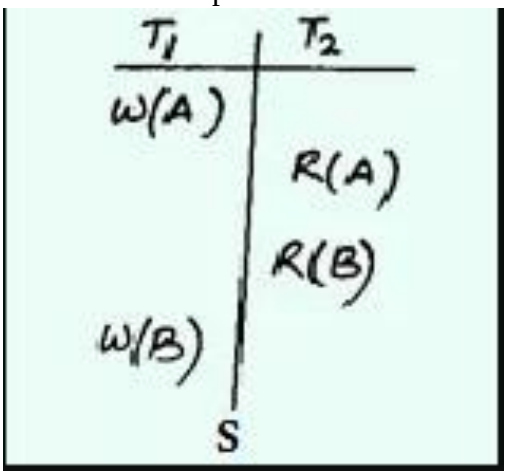
erializable schedule

If a schedule is equivalent to some serial schedule then that schedule is called Serializable schedule

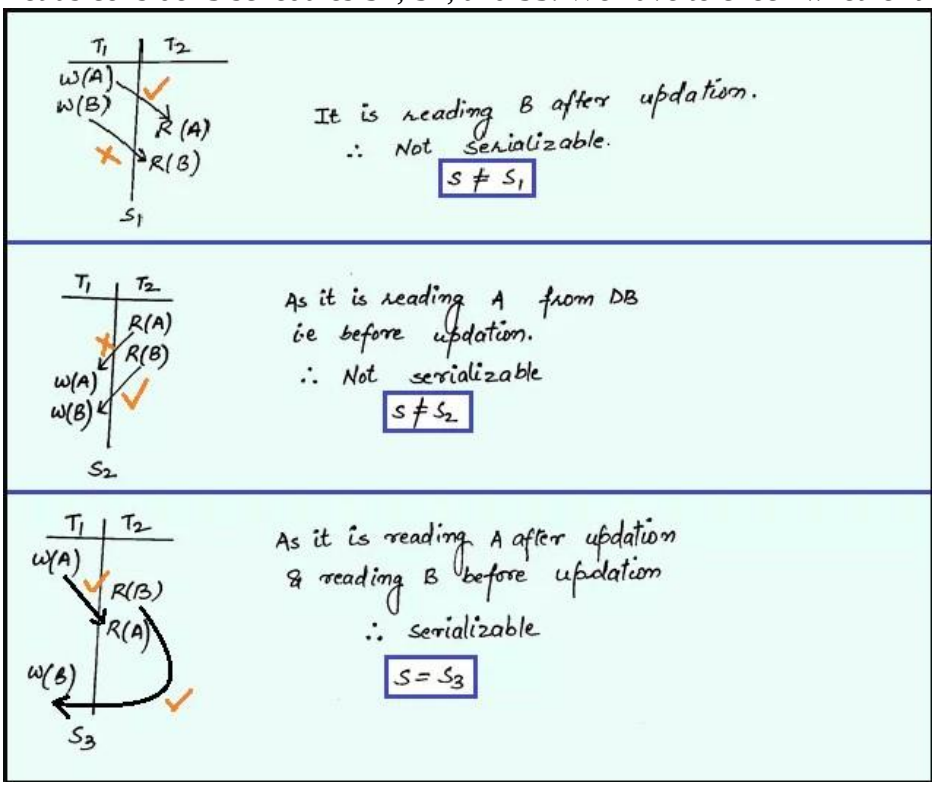
Let us consider a schedule S. What the schedule S says ?

Read A after updation.

Read B before updation.



Let us consider 3 schedules S1, S2, and S3. We have to check whether they are serializable with S or not ?



Types of Serializability

-Conflict Serializability

-View Serializability

Conflict Serializable

Any given concurrent schedule is said to be Conflict Serializable if and only if it is Conflict equivalent to one of the possible serial schedule.

Two schedules would be conflicting if they have the following properties

- Both belong to separate transactions.
- Both accesses the same data item.
- At least one of them is "write" operation.

Conflicting Instructions

Instructions l_i and l_j of transactions T_i and T_j respectively, **conflict** if they are operations by different transaction on the same data item, and at least one of these instruction is **write** operation.

1. l_i = **read**(Q), l_j = **read**(Q). l_i and l_j don't conflict.
2. l_i = **read**(Q), l_j = **write**(Q). They conflict.
3. l_i = **write**(Q), l_j = **read**(Q). They conflict
4. l_i = **write**(Q), l_j = **write**(Q). They conflict

Two schedules 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.

- 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**.
- We say that a schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule

Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable.

Schedule 3

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

Schedule 6

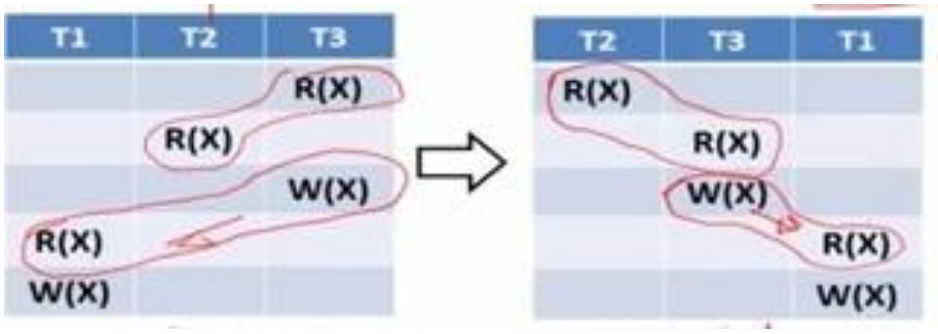
T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

View Serializable

Any given concurrent schedule is said to be View Serializable if and only if it is View equivalent to one of the possible serial schedule.

Let *S* and *S´* be two schedules with the same set of transactions. *S* and *S´* are **view equivalent** if the following three conditions are met, for each data item *Q,*

1. If in schedule S, transaction T_i reads the initial value of *Q*, then in schedule *S'* also transaction T_i must read the initial value of *Q.*
2. If in schedule S, transaction T_i executes **read**(*Q),* and that value was produced by transaction T_j (if any), then in schedule *S'* also transaction T_i must read the value of *Q* that was produced by the same **write**(Q) operation of transaction T_j .
3. The transaction (if any) that performs the final **write**(*Q*) operation in schedule *S* must also perform the final **write**(*Q*) operation in schedule *S'.*



CONCURRENCY CONTROL

Process of managing simultaneous execution of transactions in a shared database, to ensure the serializability of transactions, is known as concurrency control.

- Process of managing simultaneous operations on the database without having them interfere with one another.
- Prevents interference when two or more users are accessing database simultaneously and at least

one is updating data.

- Although two transactions may be correct in themselves, interleaving of operations may produce an incorrect result.

Why we need Concurrency Control

- Simultaneous execution of transactions over a shared database can create several data integrity and consistency problems.
- lost updated problem
- Temporary updated problem
- Incorrect summery problem

Lost updated problem

- This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.
- Successfully completed update is overridden by another user.

Example:

- T1 withdraws £10 from an account with balx, initially £100.
- T2 deposits £100 into same account.
- Serially, final balance would be £190.
- *Loss of T2's update!!*
- This can be avoided by preventing T1 from reading balx until after update.

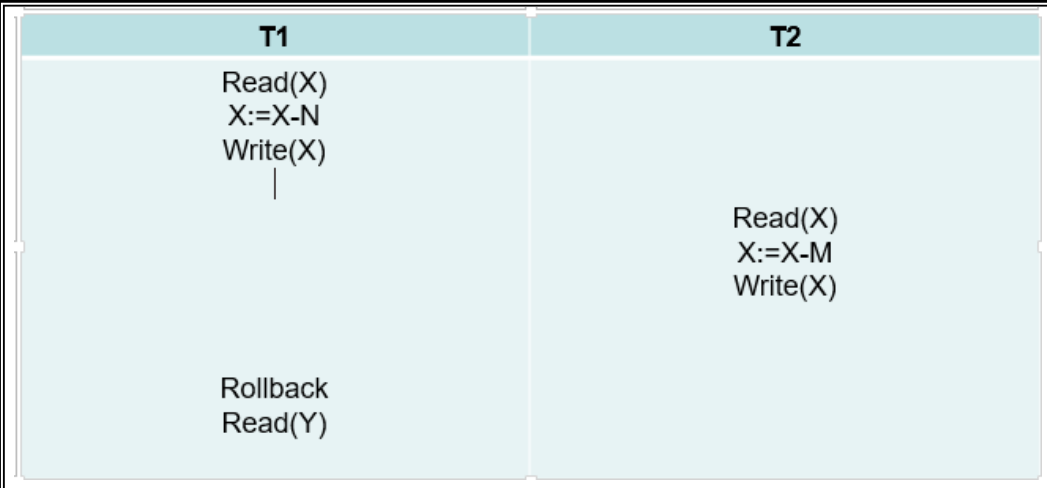
T1	T2
Read(X) X:=X-N	
	Read(X) X:=X+M
Write(X) Read(Y)	
	Write(X)
Y:=Y+N Write(Y)	

Temporary updated problem

- This problem occurs when one transaction updates a database item and then the transaction fails for some reason. The updated item is accessed by another transaction before it is changed back to its original value.
- Occurs when one transaction can see intermediate results of another transaction before it has committed.

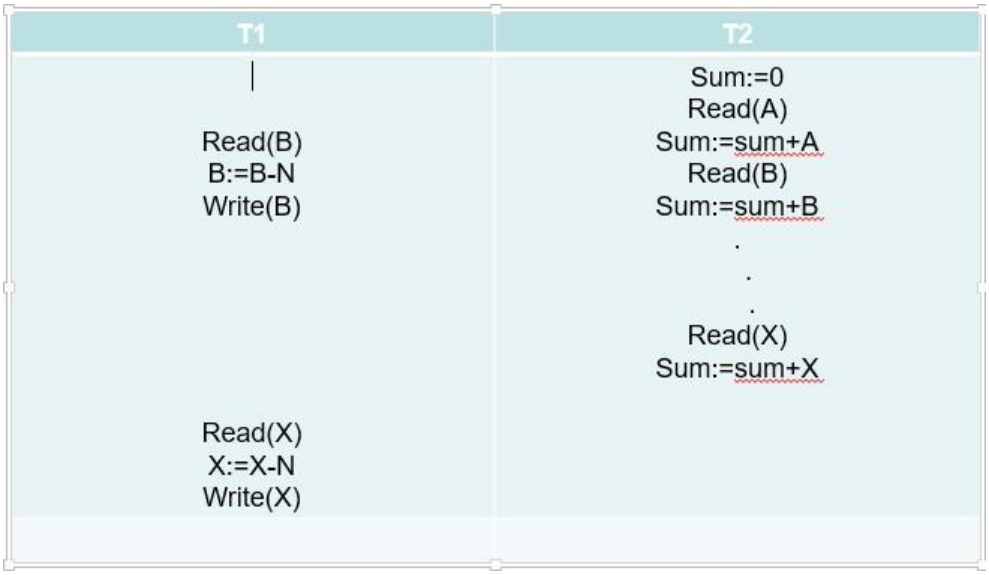
Example:

- T1 updates balx to £200 but it aborts, so balx should be back at original value of £100.
- T2 has read new value of balx (£200) and uses value as basis of £10 reduction, giving a new balance of £190, instead of £90.
- Problem avoided by preventing T2 from reading balx until after T1 commits or aborts.



Incorrect summary problem

- If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated.
- Occurs when transaction reads several values but second transaction updates some of them during execution of first.



Example:

- T6 is totaling balances of account x (£100), account y (£50), and account z (£25).
- Meantime, T5 has transferred £10 from balx to balz, so T6 now has wrong result (£10 too high).
- Problem avoided by preventing T6 from reading balx and balz until after T5 completed updates.

Concurrency control techniques

Some of the main techniques used to control the concurrent execution of transaction are based on the concept of locking the data items

LOCKING PROTOCOLS

A lock is a variable associated with a data item that describe the statues of the item with respect to possible operations that can be applied to it.

Locking is an operation which secures

(a) permission to Read

(b) permission to Write a data item for a transaction.
Example: Lock (X). Data item X is locked in behalf of the requesting transaction.
Unlocking is an operation which removes these permissions from the data item.
Example: Unlock (X): Data item X is made available to all other transactions.
Lock and Unlock are Atomic operations.
Conflict matrix

	Read	Write
Read	Y	N
Write	N	N

Lock Manager:
- Managing locks on data items.

Lock table:
- Lock manager uses it to store the identify of transaction locking a data item, the data item, lock mode and pointer to the next data item locked. One simple way to implement a lock table is through linked list

Types of lock
- Binary lock
- Read/write(shared / Exclusive) lock

Binary lock
- It can have two states (or) values 0 and 1.
  0 – unlocked
  1 - locked
- Lock value is 0 then the data item can accessed when requested.
- When the lock value is 1,the item cannot be accessed when requested.

- Lock_item(x)

```
B : if lock(x) = 0 ( * item is unlocked * )
  then lock(x)      1
            else begin
    wait ( until lock(x) = 0 )
    goto B;
  end;
```

- Unlock_item(x)

```
B : if lock(x)=1 ( * item is locked * )
    then lock(x)      0
    else
    printf (‘ already is unlocked ‘)
    goto B;
    end;
```

Read / write(shared/exclusive) lock

Read_lock

- its also called shared-mode lock
- If a transaction T_i has obtain a shared-mode lock on item X, then T_i can read, but cannot write ,X.
- Outer transactions are also allowed to read the data item but cannot write.

Read_lock(x)

```
B : if lock(x) = "unlocked"  then
                    begin
                    lock(x)        " read_locked"
                    no_of_read(x)       1
                    else if
                    lock(x) = "read_locked"
                    then
                    no_of_read(x)        no_of_read(x) +1
                    else begin
                    wait (until lock(x) = "unlocked"
                    goto B;
                    end;
```

Write_lock(x)

```
B : if lock(x) = "unlocked" then
          begin
          lock(x)       "write_locked"
          else if
          lock(x) = "write_locked"
          wait ( until lock(x) = "unlocked" )
          else begin
          lock(x)="read_locked" then
          wait ( until lock(x) = "unlocked" )
    end;
```

Unlock(x)

```
If lock(x) = "write_locked" then
Begin
Lock(x)       "unlocked"
Else if
lock(x) = "read_locked" then
Begin
No_of_read(x)        no_of_read(x) - 1
If ( no_of_read(x) = 0 ) then
Begin
Lock(x)       "unlocked"
End
```

TWO PHASE LOCKING PROTOCOL

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

- Growing phase
- Shrinking phase

Growing phase

- During this phase new locks can be occurred but none can be released

Shrinking phase

- During which existing locks can be released and no new locks can be occurred

Types

- Strict two phase locking protocol
- Rigorous two phase locking protocol

Strict two phase locking protocol

This protocol requires not only that locking be two phase, but also all exclusive locks taken by a transaction be held until that transaction commits.

Rigorous two phase locking protocol

This protocol requires that all locks be held until all transaction commits.

Consider the two transaction T_1 and T_2

T_1 : read(a_1);
read(a_2);
…….
read(a_n);
write(a_1);

T_2: read(a_1);
read(a_2);
display(a_1+a_1);

Lock conversion

- Lock Upgrade
- Lock Downgrade

Lock upgrade:

- Conversion of existing read lock to write lock
- Take place in only the growing phase

if Ti has a read-lock (X) and Tj has no read-lock (X) (i ≠ j) then
convert read-lock (X) to write-lock (X)
else
force Ti to wait until Tj unlocks X

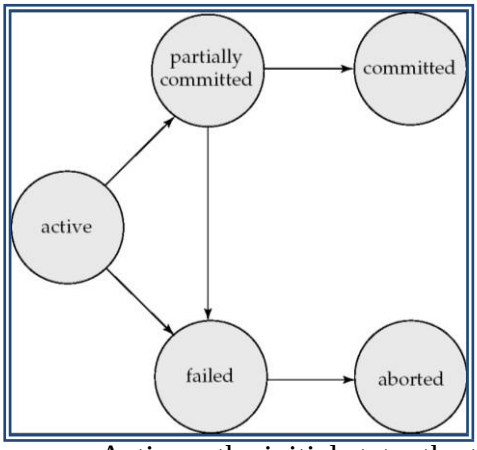
Lock downgrade:

- conversion of existing write lock to read lock
- Take place in only the shrinking phase

Ti has a write-lock (X) (*no transaction can have any lock on X*)
convert write-lock (X) to read-lock (X)

T_1	T_2
Lock-S(a_1)	
	Lock-S(a_1)
Lock-S(a_2)	
	Lock-S(a_1)
Lock-S(a_3)	
Lock-S(a_4)	
	Unlock(a_1)
	Unlock(a_2)
Lock-S(a_1)	
Upgrade(a_1)	

Transaction State



- 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 restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - restart the transaction
  - kill the transaction
- Committed – after successful completion

Log

- Log is a history of actions executed by a database management system to guarantee ACID properties over crashes or hardware failures.
- Physically, a log is a file of updates done to the database, stored in stable storage.

Log rule

- A log records for a given database update must be physically written to the log, before the update physically written to the database.
- All other log record for a given transaction must be physically written to the log, before the commit log record for the transaction is physically written to the log.
- Commit processing for a given transaction must not complete until the commit log record for the transaction is physically written to the log.

System log

- **[Begin transaction ,T]**
- **[write_item , T, X , oldvalue,newvalue]**
- **[read_item,T,X]**
- **[commit,T]**
- **[abort,T]**

TWO - PHASE COMMIT

- Assumes fail-stop model – failed sites simply stop working, and do not cause any other harm, such as sending incorrect messages to other sites.
- Execution of the protocol is initiated by the coordinator after the last step of the transaction has been reached.
- The protocol involves all the local sites at which the transaction executed
- Let T be a transaction initiated at site S_i, and let the transaction coordinator at S_i be C_i

Phase 1: Obtaining a Decision (prepare)

- Coordinator asks all participants to *prepare* to commit transaction T_i.

  - C_i adds the records <prepare *T*> to the log and forces log to stable storage
  - sends prepare *T* messages to all sites at which *T* executed
- Upon receiving message, transaction manager at site determines if it can commit the transaction
  - if not, add a record <no *T*> to the log and send abort *T* message to C_i
  - if the transaction can be committed, then:
  - add the record <ready *T*> to the log
  - force *all records* for *T* to stable storage
  - send ready *T* message to C_i

Phase 2: Recording the Decision (commit)

- *T* can be committed of C_i received a ready *T* message from all the participating sites: otherwise *T* must be aborted.
- Coordinator adds a decision record, <commit *T*> or <abort *T*>, to the log and forces record onto stable storage. Once the record stable storage it is irrevocable (even if failures occur)
- Coordinator sends a message to each participant informing it of the decision (commit or abort)
- Participants take appropriate action locally.

Handling of Failures - Site Failure

When site S_i recovers, it examines its log to determine the fate of transactions active at the time of the failure.

- Log contain <commit *T*> record: site executes redo (*T*)
- Log contains <abort *T*> record: site executes undo (*T*)
- Log contains <ready *T*> record: site must consult C_i to determine the fate of *T*.
  - If *T* committed, redo (*T*)
  - If *T* aborted, undo (*T*)
- The log contains no control records concerning *T* replies that S_k failed before responding to the prepare *T* message from C_i
  - since the failure of S_k precludes the sending of such a response C_1 must abort *T*
  - S_k must execute undo (*T*)

Handling of Failures- Coordinator Failure

- If coordinator fails while the commit protocol for *T* is executing then participating sites must decide on *T*'s fate:
  1. If an active site contains a <commit *T*> record in its log, then *T* must be committed.
  2. If an active site contains an <abort *T*> record in its log, then *T* must be aborted.
  3. If some active participating site does not contain a <ready *T*> record in its log, then the failed coordinator C_i cannot have decided to commit *T*. Can therefore abort *T*.
  4. If none of the above cases holds, then all active sites must have a <ready *T*> record in their logs, but no additional control records (such as <abort *T*> of <commit *T*>). In this case active sites must wait for C_i to recover, to find decision.
- Blocking problem : active sites may have to wait for failed coordinator to recover.

Handling of Failures - Network Partition

- If the coordinator and all its participants remain in one partition, the failure has no effect on the commit protocol.
- If the coordinator and its participants belong to several partitions:
  - Sites that are not in the partition containing the coordinator think the coordinator has failed, and execute the protocol to deal with failure of the coordinator.

- No harm results, but sites may still have to wait for decision from coordinator.

- The coordinator and the sites are in the same partition as the coordinator think that the sites in the other partition have failed, and follow the usual commit protocol.
  - Again, no harm results

DEADLOCK

System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.

Consider the following two transactions:

T_1: write (*A*) T_2: write(*A*)
write(*B*) write(*B*)

Schedule with deadlock

T_1	T_2
lock-X on *A* write (*A*) wait for **lock-X** on *B*	 **lock-X** on *B* write (*B*) wait for **lock-X** on *A*

Deadlock Handling

Deadlock prevention protocol

Ensure that the system will *never* enter into a deadlock state.

Some prevention strategies :

Approach1

- Require that each transaction locks all its data items before it begins execution either all are locked in one step or none are locked.
- Disadvantages
  - Hard to predict ,before transaction begins, what data item need to be locked.
  - Data item utilization may be very low.

Approach2

- Assign a unique timestamp to each transaction.
- These timestamps only to decide whether a transaction should wait or rollback.

schemes:

- wait-die scheme
- wound-wait scheme

wait-die scheme

- non preemptive technique

When transaction Ti request a data item currently held by Tj, Ti is allowed to wait only if it has a timestamp smaller than that of Tj. otherwise ,Ti rolled back(dies)

- **older transaction may wait for younger one** to release data item. Younger transactions

never wait for older ones; they are rolled back instead.

– A transaction may die several times before acquiring needed data item

Example.

- Transaction T_1,T_2,T_3 have time stamps 5,10,15,respectively.
- if T_1 requests a data item held by T_2,then T_1 will wait.
- If T_3 request a data item held by T_2,then T_3 will be rolled back.

.wound-wait scheme

- Preemptive technique

- When transaction T_i requests a data item currently held by T_j,T_i is allowed to wait only if it has a timestamp larger than that of T_j. Otherwise T_j is rolled back

– Older transaction wounds (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.

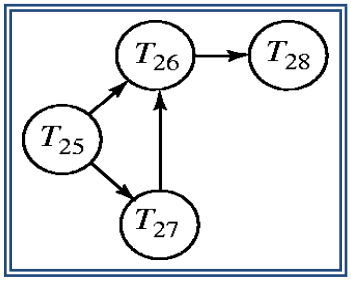
Example

- Transaction T_1,T_2,T_3 have time stamps 5,10,15,respectively.
- if T_1 requests a data item held by T_2,then the data item will be preempted from T_2,and T_2 will be rolled back.
- If T_3 requests a data item held by T_2,then T_3 will wait.

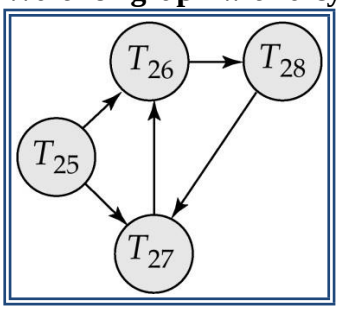
Deadlock Detection

- Deadlocks can be described as a *wait-for graph*, which consists of a pair $G = (V,E)$,
  - V is a set of vertices
  - E is a set of edges
- If $T_i \rightarrow T_j$ is in E, then there is a directed edge from T_i to T_j, implying that T_i is waiting for T_j to release a data item.
- The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.

Wait-for graph without a cycle



Wait-for graph with a cycle



Recovery from deadlock

- The common solution is to roll back one or more transactions to break the deadlock.
- Three action need to be taken
  - Selection of victim

- Rollback
- Starvation

Selection of victim

- Set of deadlocked transations,must determine which transaction to roll back to break the deadlock.
- Consider the factor minimum cost

Rollback

- once we decided that a particular transaction must be rolled back, must determine how far this transaction should be rolled back
- Total rollback
- Partial rollback

Starvation

Ensure that a transaction can be picked as victim only a finite number of times.

Intent locking

- Intent locks are put on all the ancestors of a node before that node is locked explicitly.
- If a node is locked in an intention mode, explicit locking is being done at a lower level of the tree.

Types of Intent Locking

- Intent shared lock(IS)
- Intent exclusive lock(IX)
- Shared lock (S)
- Shared Intent exclusive lock (SIX)
- Exclusive lock (X)

Intent shared lock(IS)

- If a node is locked in indent shared mode, explicit locking is being done at a lower level of the tree, but with only shared-mode lock
- Suppose the transaction T_1 reads record r_{a2}in file $F_{a.}$ Then,T_1 needs to lock the database, area $A_{1,and}$ F_a in IS mode, and finally lock r_{a2} in S mode.

Intent exclusive lock(IX)

If a node is locked in intent locking is being done at a lower level of the tree, but with exclusive mode or shared-mode locks.

- Suppose the transaction T_2 modifies record r_{a9} in file $F_{a.}$ Then,T_2 needs to lock the database, area $A_{1,and}$ F_a in IX mode, and finally to lock r_{a9} in X mode.

Shared Intent exclusive lock (SIX)

If the node is locked in Shared Intent exclusive mode, the subtree rooted by that node is locked explicitly in shared mode, and that explicit locking is being done at lower level with exclusive mode.

Shared lock (S)

-T can tolerate concurrent readers but not concurrent updaters in R.

Exclusive lock (X)

-T cannot tolerate any concurrent access to R at all.

Lock compatibility

Tran 2

Tran 1	NL	IS	IX	S	SIX	X
NL	Yes	Yes	Yes	Yes	Yes	Yes
IS	Yes	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	Yes	No	No	No
S	Yes	Yes	No	Yes	No	No
SIX	Yes	Yes	No	No	No	No
X	Yes	No	No	No	No	No

If Tran 1 holds a lock of the given type and Tran 2 requests another lock of the given type will that request be granted?

TRANSACTION RECOVERY

Recovery Algorithms

- Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures
- Recovery algorithms have two parts
    1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures
    2. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

Example

```
Begin transaction
Update Acc 1001{balance:=Balance-100};
If any error occurred then
Goto Undo;
End if;
Update Acc 1002{balance:=balance+100};
If any error occurred then
Goto undo;
End if;
Commit;
Goto finish;
Undo: rollback;
Finish:  return;
```

Requirement for recovery

- Implicit rollback
- Message handling
- Recovery log
- Statement atomicity
- No nested transaction

Transaction recovery

Database updates are kept in buffer in main memory and not physically written to disk until commit**.**

System recovery

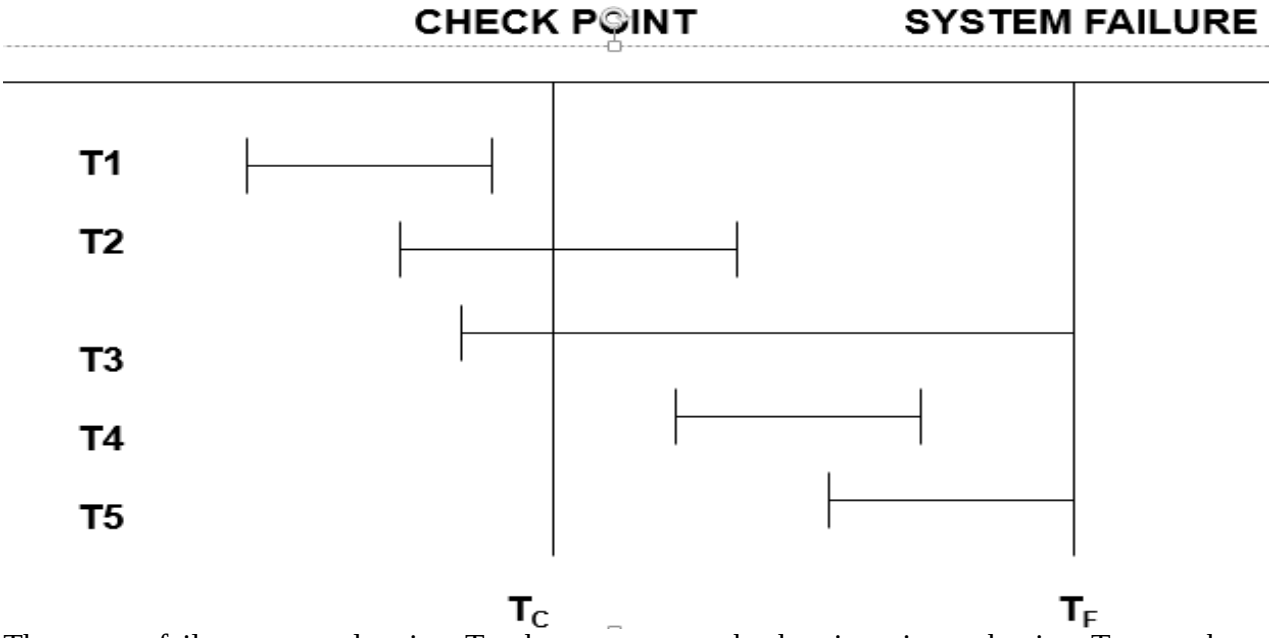
Local failures –affect only the transaction which the failure has actually occurred.

Global failures- affect all the transaction in progress at the time of failure.

System failure – do not physically damage the DB Eg: power shut down
Media failure-cause damage to the DB. Eg: head crash

ARIES Recovery Algorithm

- ARIES-Algorithm for Recovery and Isolation Exploiting Semantics
- ARIES recovery involves three passes
  Analysis pass: Determines the REDO and UNDO lists.
  Redo pass: Repeats history, redoing all actions from REDO List
  Undo pass: Rolls back all incomplete transactions



- The system failure occurred at time T_f , the most recent check point prior to the time T_f was taken at a time T_f
- Start with two list of transaction the UNDO and REDO list
- search forward through the log starting from check point.
- if begin transaction log record is found for transaction(T) add T to UNDO list.
- if commit log record is found for transaction(T),add T to REDO list
- when the end of log record is reached the UNDO and REDO list is identified

UNDO	REDO
T_3	T_2
T_5	T_4

SAVE POINTS

- It is possible for a transaction to create a savepoint.
- It is used to store intermediate results

So that it will rollback to a previously established savepoint whenever any recovery process starts.
Create: Savepoint <savepoint_name>;
Rollback: Rollback to <savepoint_name>;
Drop: Release <savepoint_name>;

SQL

COMMIT: Used to made the changes permanently in the Database.
SAVEPOINT: Used to create a savepoint or a reference point.

ROLLBACK: Similar to the undo operation.

Example:

```
SQL> select * from customer;
   CUSTID    PID       QUANTITY
    ----------  ----------  ---------
      100       1234        10
      101       1235        15
      102       1236        15
      103       1237        10
SQL> savepoint s1;
Savepoint created.
SQL> Delete from customer where custid=103;
   CUSTID   PID       QUANTITY
    ----------  ---------  ---------
      100       1234        10
      101       1235        15
      102       1236        15
SQL> rollback to s1;
Rollback complete.
SQL> select * from customer;
   CUSTID   PID      QUANTITY
   ----------  ------------------
      100       1234        10
      101       1235        15
      102       1236        15
      103       1237        10
SQL> commit;
```

ISOLATION LEVEL

- Degree of interference
- An isolation levels mechanism is used to isolate each transaction in a multi-user environment
- **Dirty Reads:** This situation occurs when transactions read data that has not been committed.
- **Nonrepeatable Reads:** This situation occurs when a transaction reads the same query multiple times and results are not the same each time
- **Phantoms:** This situation occurs when a row of data matches the first time but does not match subsequent times

Types

Higher isolation level (Repeatable read)

- Less interference
- Lower concurrency
- All schedules are serializable

Lower isolation level(cursor stability)

- More interference
- Higher concurrency
- Not a serializable

One special problem that can occur if transaction operates at less than the maximum isolation level (i.e) less then repeatable read level is called phantom problem.