TRANSACTION PROCESSING

Transaction concurrency control
Serilazibility
locking techniques
Time stamping techniques
Two phase locking system
Deadlock handling

What is a transaction

- A unit of program execution that accesses and possibly updates various data items.
- A transaction is the basic logical unit of execution in an information system.
- A transaction is a sequence of operations that must be executed as a whole, taking a consistent (& correct) database state into another consistent (& correct) database state.
- A collection of actions that make consistent transformations of system states while preserving system consistency
- An indivisible unit of processing

database in a
consistent state

database in a
consistent state

begin Transaction

database may be
temporarily in an
inconsistent state
during execution

end Transaction

ACID Transactions

- **A** ***Atomicity***: a transaction is an atomic unit of processing and it is either performed entirely or not at all
- **C** ***Consistency Preservation***: a transaction's correct execution must take the database from one correct state to another
- **I** ***Isolation/Independence***: the updates of a transaction must not be made visible to other transactions until it is committed (solves the temporary update problem).Each transaction is unaware of other transactions executing concurrently in the system.
- **D** ***Durability*** (or Permanency): if a transaction changes the database and is committed, the changes must never be lost because of subsequent failure

Serializability

- **Serializability** ensures that the schedule for the concurrent execution of the transactions yields consistent results. This property is important in multiuser and distributed databases, where multiple transactions are likely to be executed concurrently

Example

Consider a simplified banking system consisting of several accounts and set of transactions that access and update those accounts.

Access to database is accomplished by following two operations.

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

Write operation may be temporarily stored in memory and executed on the disk later.

Let Ti be a transaction that transfers Rs. 500 from account A to account B.

```
Ti:     read(A)
        A:=A-500
        write(A)
        read(B)
        B:=B+500
        write(B)
```

Consistency

- Consistency requirement is that the sum of A and B be unchanged by the execution of the transaction.
- If the database is consistent before an execution of the transaction, the database remains consistent after the execution of the transaction.
- Ensuring consistency is the responsiblity of application programmer.
- Automatic testing of integrity constraints may facilitate this task.

Atomicity

- Prior to the execution of transaction Ti the values of A and B are Rs. 1000 and Rs.2000 resp.
- Consider that during the execution of Ti a failure occurred after the write(A) operation executed and before the write(B) executed.
- The values of A and B will be Rs.950 and Rs. 2000.
- A+B not true.
- That is the reason for atomicity requirement.
- If the atomicity property is provided all actions of the transaction are reflected in the database or none.
- The DB system keeps track of the old values of any data on which a transaction performs a write and if the transaction doesn't complete its execution the old values are restored to make it appear as though the transaction never executed.

Durability

- Guarantees that once a transaction completes successfully, all the updates that it carries out on the database persists, even if there is a system failure after the transaction completes execution.
- The updates carried out by the transaction completes execution.

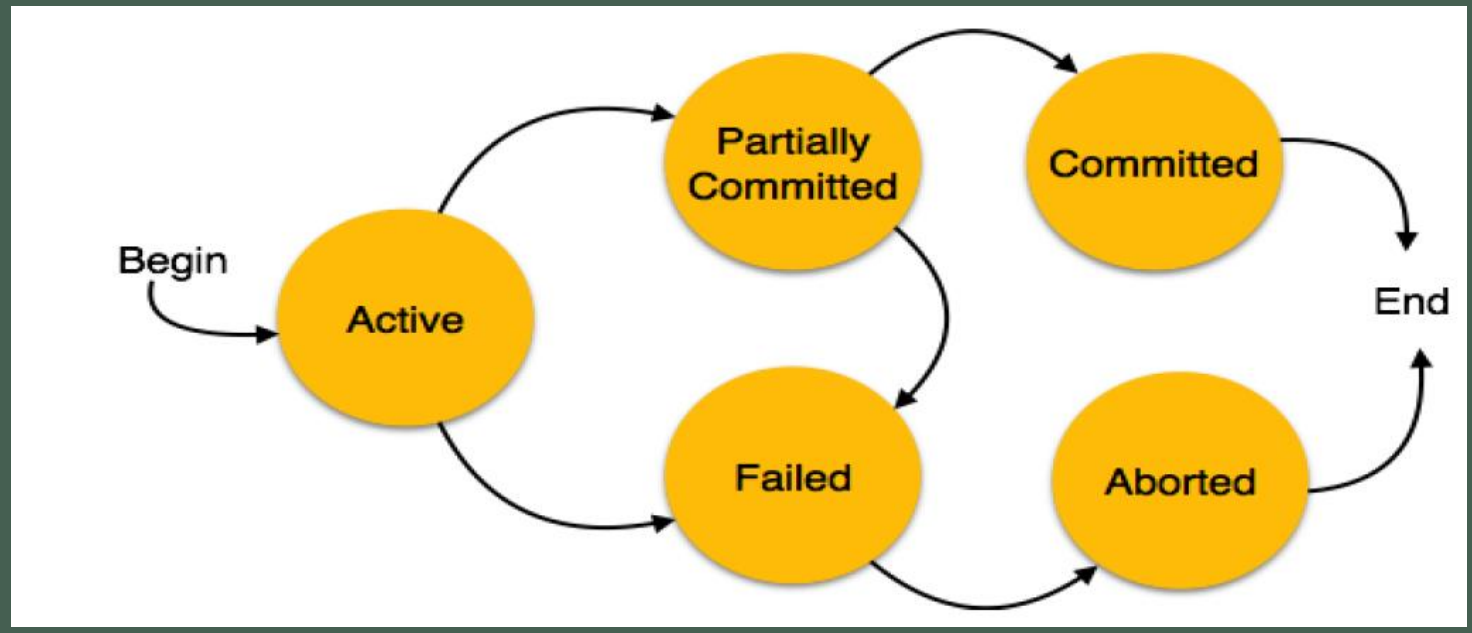
Isolation

- Ensures that the concurrent execution of transactions results in a system state that is equivalent to the state that could have been obtained had these transactions executed one at a time in some order.
- Eg: if any second transaction read the value of A and B in between and computes A+B , it will observe an inconsistent value.

TRANSACTION STATES

- Active
  - 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 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.

TRANSACTION STATES



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

- The database system then writes out enough information to disk that, even in the event of a failure, the updates performed by the transaction can be re-created when the system restarts after the failure.
- When the last of this information is written out, the transaction enters the committed state.

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

- At this point, the system has two options:
- It can restart the transaction, but only if the transaction was aborted as a result of some hardware or software error that was not created through the internal logic of the transaction. A restarted transaction is considered to be a new transaction.
- It can kill the transaction. It usually does so because of some internal logical error that can be corrected only by rewriting the application program, or because the input was bad, or because the desired data were not found in the database.

Concurrency control

- Transaction-processing systems usually allow multiple transactions to run concurrently.
- Allowing multiple transactions to update data concurrently causes several complications with consistency of the data.
- 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.
- good reasons for allowing concurrency:
- **Improved throughput and resource utilization**.
- **Reduced waiting time**

The database system controls the interaction among the concurrent transactions to prevent them from destroying the consistency of the database through **concurrency-control schemes**.

Consider the simplified banking system which has several accounts, and a set of transactions that access and update those accounts.

Let *T*1 and *T*2 be two transactions that transfer funds from one account to another.

Transaction *T*1 transfers ₹ 50 from account *A* to account *B*. It is defined as

```
T1: read(A);
A := A − 50;
write(A);
read(B);
B := B + 50;
write(B).
```

Transaction *T*2 transfers 10 percent of the balance from account *A* to account *B*. It is defined as

```
T2: read(A);
temp := A * 0.1;
A := A − temp;
write(A);
read(B);
B := B + temp;
write(B).
```

- Suppose the current values of accounts *A* and *B* are ₹1000 and ₹ 2000, respectively.
- Suppose that the two transactions are executed one at a time in the order *T*1 followed by *T*2.
- The final values of accounts *A* and *B*, after the execution takes place, are ₹ 855 and ₹ 2145, respectively. Thus, the total amount of money in accounts *A* and *B*—that is, the sum *A* + *B*—is preserved after the execution of both transactions.

*T*1
read(*A*)
A := *A* – 50
write (*A*)
read(*B*)
B := *B* + 50
write(*B*)

*T*2
read(*A*)
temp := *A* * 0.1
A := *A* – *temp*
write(*A*)
read(*B*)
B := *B* + *temp*
write(*B*)

If the transactions are executed one at a time in the order *T*2 followed by *T*1, then again, the sum *A* + *B* is preserved, and the final values of accounts *A* and *B* are ₹850 and ₹ 2150, respectively. The execution sequences just described are called **schedules**

T_1
read(*A*)
temp := *A* * 0.1
A := *A* – *temp*
write(*A*)
read(*B*)
B := *B* + *temp*
write(*B*)

T_2

read(*A*)
A := *A* 50
write(*A*)
read(*B*)
B := *B* + 50
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. These schedules are serial.

- When the database system executes several transactions concurrently, the corresponding schedule no longer needs to be serial.
- If two transactions are running concurrently, the operating system may execute one transaction for a little while, then perform a context switch, execute the second transaction for some time, and then switch back to the first transaction for some time, and so on.
- With multiple transactions, the CPU time is shared among all the transactions.
- Several execution sequences are possible, since the various instructions from both transactions may now be interleaved.
- it is not possible to predict exactly how many instructions of a transaction will be executed before the CPU switches to another transaction.
- Thus, the number of possible schedules for a set of *n* transactions is much larger than *n*!.

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)

We can ensure consistency of the database under concurrent execution by making sure that any schedule that executed has the same effect as a schedule that could have occurred without any concurrent execution. That is, the schedule should, in some sense, be equivalent to a serial schedule.

Serializability

- The database system must control concurrent execution of transactions, to ensure that the database state remains consistent.

*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*)

The only significant operations of a transaction, from a scheduling point of view, are its read and write instructions.

Lock-Based Protocols

- To ensure serializability is to require that data items be accessed in a mutually exclusive manner(while one transaction is accessing a data item, no other transaction can modify that data item.)
- The most common method used to implement this requirement is to allow a transaction to access a data item only if it is currently holding a lock on that item

Locks

Various modes in which a data item may be locked are:

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

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

Lock-compatibility matrix comp

	S	X
S	true	false
X	false	false

- Every transaction **request** a lock in an appropriate mode on data item Q, depending on the types of operations that it will perform on Q.
- The transaction makes the request to the concurrency-control manager.
- The transaction can proceed with the operation only after the concurrency-control manager **grants** the lock to the transaction.
- Given a set of lock modes, a **compatibility function** is defined on them as follows.
- Let *A* and *B* represent arbitrary lock modes.
- Suppose that a transaction *Ti* requests a lock of mode *A* on item Q on which transaction *Tj* (*Ti != Tj*) currently holds a lock of mode *B*.
- If transaction *Ti* can be granted a lock on Q immediately, in spite of the presence of the mode *B* lock, then we say mode *A* is **compatible** with mode *B*.
- Such a function can be represented conveniently by a matrix.
- An element comp(*A, B*) of the matrix has the value *true* if and only if mode *A* is compatible with mode *B*.

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

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

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

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

consider the simplified banking system. Let *A* and *B* be two accounts that are accessed by transactions *T*1 and *T*2. Transaction *T*1 transfers Rs. 50 from account *B* to account *A*. Transaction *T*2 displays the total amount of money in accounts *A* and *B* ,the sum *A* + *B*

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

Figure 16.4 Schedule 1.

- Suppose a transaction *T*2 has a shared-mode lock on a data item, and another transaction *T*1 requests an exclusive-mode lock on the data item.
- *T*1 has to wait for *T*2 to release the shared-mode lock.
- a transaction *T*3 may request a shared-mode lock on the same data item.
- The lock request is compatible with the lock granted to *T*2, so *T*3 may be granted the shared-mode lock.
- *T*2 may release the lock, but still *T*1 has to wait for *T*3 to finish.
- a new transaction *T*4 that requests a shared-mode lock on the same data item, and is granted the lock before *T*3 releases it.
- there is a sequence of transactions that each requests a shared-mode lock on the data item, and each transaction releases the lock a short while after it is granted, but *T*1 never gets the exclusive-mode lock on the data item.
- The transaction *T*1 may never make progress, and is said to be **starved**.

- Starvation of transactions can be avoided by granting locks in the following manner:
- When a transaction *Ti* requests a lock on a data item *Q* in a particular mode *M*, the concurrency-control manager grants the lock provided that
  - There is no other transaction holding a lock on *Q* in a mode that conflicts with *M*.
  - There is no other transaction that is waiting for a lock on *Q*, and that made its lock request before *Ti*.
- Thus, a lock request will never get blocked by a lock request that is made later.

Two-Phase Locking Protocol

- It ensures serializability.
- Each transaction issue lock and unlock requests in two phases:
  1. **Growing phase**. A transaction may obtain locks, but may not release any lock.
  2. **Shrinking phase**. A transaction may release locks, but may not obtain any new locks.

Two-Phase Locking Protocol

- Initially, a transaction is in the growing phase.
- The transaction acquires locks as needed.
- Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests.
- The two-phase locking protocol ensures conflict serializability.
- The point in the schedule where the transaction has obtained its final lock (the end of its growing phase) is called the **lock point** of the transaction.
- Two-phase locking does *not* ensure freedom from deadlock

Timestamp-Based Protocols

Timestamps

- With each transaction *Ti* in the system, a unique fixed timestamp is associated , denoted by TS(*Ti*).
- It 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*).
- Methods for implementing this scheme:
  - Use the value of the *system clock* as the timestamp;
  - Use a **logical counter** that is incremented after a new timestamp has been assigned; when the transaction enters the system.

- The timestamps of the transactions determine the serializability order.
  - If TS(*Ti*) < TS(*Tj*), then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction *Ti* appears before transaction *Tj* .
  - To implement this scheme, we associate with each data item *Q* two timestamp values:
  - **W-timestamp(*Q*)** denotes the largest timestamp of any transaction that executed write(*Q*) successfully.
  - **R-timestamp(Q)** denotes the largest timestamp of any transaction that executed read(*Q*) successfully.
  - Timestamps are updated whenever a new read(*Q*) or write(*Q*) instruction is executed.

Deadlock Handling

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.

If there exists a set of waiting transactions {T0, T1, . . ., Tn} such that T0 is waiting for a data 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, then none of the transactions can make progress.

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

- **deadlock prevention**
- **deadlock detection** and **deadlock recovery**

Deadlock Prevention

Two approaches to deadlock prevention.

- ✓ Ensure that no cyclic waits can occur by ordering the requests for locks, or requiring all locks to be acquired together.
- ✓ Perform transaction rollback instead of waiting for a lock, whenever the wait could potentially result in a deadlock.

- First approach requires that each transaction locks all its data items before it begins execution.
- Either all are locked in one step or none are locked.

Two main **disadvantages** to this protocol:

(1) it is often 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.

- The second approach is to use preemption and transaction rollbacks.
- **Preemption:** when a transaction *T*2 requests a lock that transaction *T*1 holds, the lock granted to *T*1 may be **preempted** by rolling back of *T*1, and granting of the lock to *T*2.
- **Assign a unique timestamp to each transaction.**
- These timestamps are used to decide whether a transaction should wait or roll back. If a transaction is rolled back, it retains its *old* timestamp when restarted.

Two different deadlock prevention schemes using timestamps are:

- **wait–die** scheme
  - ✓ *The **wait–die** scheme is a nonpreemptive 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 *T*2, *T*3, and *T*4 have timestamps 5, 10, and 15, respectively. If *T*2 requests a data item held by *T*3, then *T*2 will wait. If *T*4 requests a data item held by *T*3, then *T*4 will be rolled back.

➢ **wound–wait** scheme

- ✓ The **wound–wait** scheme is a preemptive technique.
- ✓ 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*).

Transactions *T*2, *T*3, and *T*4, if *T*2 requests a data item held by *T*3, then the data item will be preempted from *T*3, and *T*3 will be rolled back. If *T*4 requests a data item held by *T*3, then *T*4 will wait.

- In the **wait–die** scheme, an older transaction must wait for a younger one to release its data item. Thus, the older the transaction gets, the more it tends to wait.
- In the wound–wait scheme, an older transaction never waits for a younger transaction.
- In the **wait–die scheme**, if a transaction *Ti* dies and is rolled back because it requested a data item held by transaction *Tj*, then *Ti* may reissue the same sequence of requests when it is restarted. If the data item is still held by *Tj* , then *Ti* will die again. Thus, *Ti* may die several times before acquiring the needed data item.
- In the **wound–wait scheme**, Transaction *Ti* is wounded and rolled back because *Tj* requested a data item that it holds. When *Ti* is restarted and requests the data item now being held by *Tj* , *Ti* waits. Thus, there may be fewer rollbacks in the wound–wait scheme.

Timeout-Based Schemes

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 timeout, and it rolls itself back and restarts.

Deadlock

An algorithm that examines the state of the system is invoked periodically to determine whether a deadlock has occurred. If one has, then the system must attempt to recover from the deadlock.

To do so, the system must:

- Maintain information about the current allocation of data items to transactions, as well as any outstanding data item requests.
- Provide an algorithm that uses this information to determine whether the system has entered a deadlock state.
- Recover from the deadlock when the detection algorithm determines that a deadlock exists.

Deadlock Detection

- **The wait-for 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 $T_i \rightarrow T_j$.
- If $T_i \rightarrow T_j$ is in E, then there is a directed edge from transaction T_i to T_j , implying that transaction T_i i waiting for transaction T_j to release a data item that it needs.
- When transaction T_i requests a data item currently being held by transaction T_j , then the edge $T_i \rightarrow T_j$ is inserted in the wait-for graph.
- This edge is removed only when transaction T_j is no longer holding a data item needed by transaction T_i.
- 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

- Transaction *T*25 is waiting for transactions *T*26 and *T*27.
- Transaction *T*27 is waiting for transaction *T*26.
- Transaction *T*26 is waiting for transaction *T*28.

Since the graph has no cycle, the system is not in a deadlock state

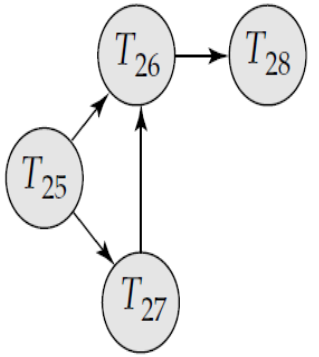


Figure 16.18 Wait-for graph with no cycle.

- *T*28 is requesting an item held by *T*27.
- The edge *T*28 → *T*27 is added to the wait-for graph
- The graph contains the cycle *T*26 → *T*28 →*T*27 →*T*26
- implying that transactions *T*26, *T*27, and *T*28 are all deadlocked.

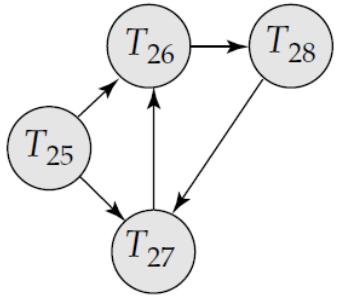


Figure 16.19 Wait-for graph with a cycle.

Recovery from Deadlock

- When a detection algorithm determines that a deadlock exists, the system must **recover** from the deadlock.
- Roll back one or more transactions to break the deadlock.
- Three actions need to be taken:
- **Selection of a victim**
  - Given a set of deadlocked transactions, determine which transaction (or transactions) to roll back to break the deadlock.
  - Roll back those transactions that will incur the minimum cost.
  - Factors may determine the cost of a rollback, including
    - How long the transaction has computed, and how much longer the transaction will compute before it completes its designated task.
    - How many data items the transaction has used.
    - How many more data items the transaction needs for it to complete.
    - How many transactions will be involved in the rollback.

- **Rollback**.
  - Once we have decided that a particular transaction must be rolled back, we must determine how far this transaction should be rolled back.
    - **total rollback**: Abort the transaction and then restart it.
    - **partial rollback**
      - requires the system to maintain additional information about the state of all the running transactions.
      - The deadlock detection mechanism should decide which locks the selected transaction needs to release in order to break the deadlock.
      - The selected transaction must be rolled back to the point where it obtained the first of these locks, undoing all actions it took after that point.
      - The recovery mechanism must be capable of performing such partial rollbacks. The transactions must be capable of resuming execution after a partial rollback.

- **Starvation**.
  - the same transaction is always rollbacked.
  - This transaction never completes its designated task, thus there is **starvation**.
  - 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.

Thank You!!!