Unit 8: Concurrency Control

- The technique used to protect data when multiple users (translations) are accessing (using) the same data concurrently (at same time) is called Concurrency Control.
- Concurrency control techniques aim to achieve serializability and recoverability.
- Different concurrency control schemes can be used to ensure the isolation property is ensured when multiple transactions are executed in parallel.
- The DBMS must guarantee that only serializable recoverable schedules are generated and it also guarantees that no effect of committed transaction is lost, and no effect of aborted (roll back) transaction remains in the related database.
- Due to concurrent execution it is very hard to preserve the isolation property. To ensure it, the system must control the interaction among the concurrent transactions.
- The control is achieved through one of the mechanism called concurrency control schemes.
- Most high performance transactional systems need to run transactions concurrently to meet their performance requirements. Thus, without concurrency control such systems can neither provide correct results nor maintain their database consistent.
- Till now we already know how to check whether a schedule will maintain the consistency of the database or not using conflict serializability and view serializability, and how to check whether schedule is recoverable or cascadeless or strict schedule. Now we will study protocols that guarantee to generate schedules which satisfy these properties.

Lock Based Protocol

- Major problem in the database is concurrency.
- Concurrency problems arries when multiple users try to update or insert data into the database table at the same time. Such concurrent updates can cause data to become corrupt or inconsistent.
- Locking is a strategy that is used to prevent such concurrent updates to data.
- A lock is a mechanism to control concurrent access to a data item.

Simple Locking Protocol/ Shared-Exclusive Locking Protocol

- Data items can be locked in two modes :
    1. exclusive (X) mode: Data item can be both read as well as written. X-lock is requested using lock-X instruction. Denoted as lock-X(A) or X(A).
    2. shared (S) mode: Data item can only be read. S-lock is requested using lock-S instruction. Denoted as lock-S(A) or S(A).
- Lock requests are made to the concurrency-control manager. Transactions can proceed only after a request is granted.

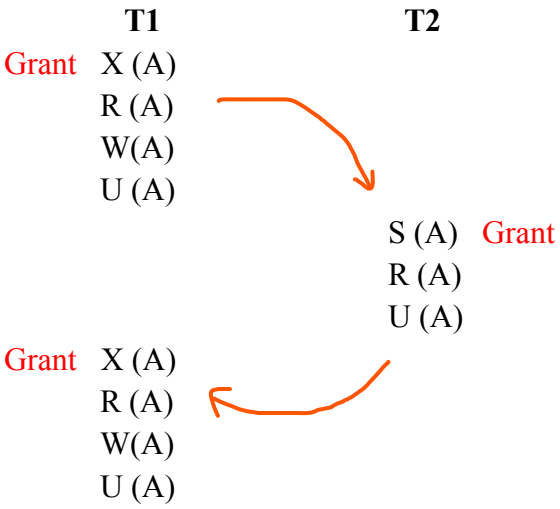
Lock compatibility matrix

	S	**X**
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item, but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.
- Example of a transaction performing locking:

```
T 2 :   lock-S(A);
        read (A);
        unlock(A);
        lock-S(B);
        read (B);
        unlock(B);
        display(A+B);
```

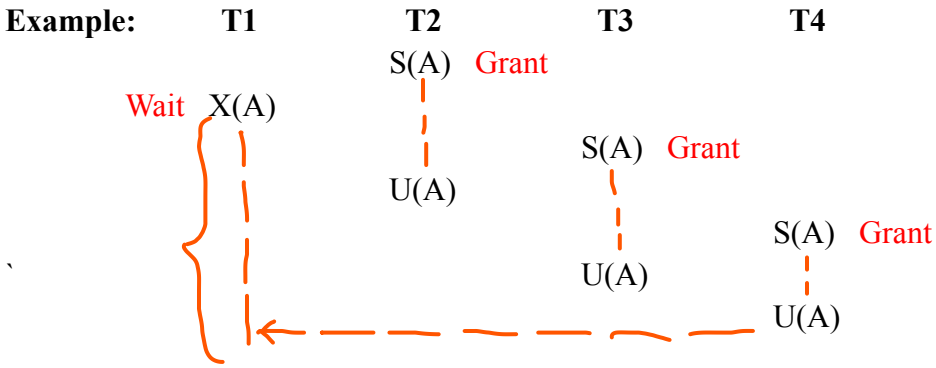
- Locking as above is not sufficient to guarantee serializability — if A and B get updated in-between the read of A and B, the displayed sum would be wrong.
- Example of a transaction performing locking: non-serial schedule example



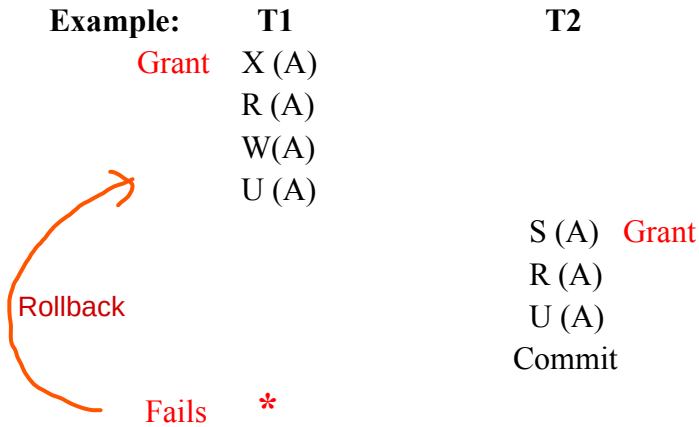
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

- The locking protocol must ensure serializability.
- The locking protocol may not free from deadlock.

- The locking protocol may not free from starvation.



- The locking protocol may not free from irrecoverability.



Sample Transactions with Locks

T1: lock-X(B)	T2: lock-S(A)
read(B)	read(A)
B = B -50;	unlock(A)
write(B);	lock-S(B)
unlock(B);	read(B);
lock-X(A);	unlock(B);
read(A);	display(A+B);
A = A + 50;	
write(A);	
unlock(A);	

Schedule for Transactions: non-serializable schedule

T1	T2	concurrency-control manager
lock-X(B)		
		grant-X(B, T1)
Read(B		
B = B - 50;		
write(B);		
unlock(B);		
	lock-S(A)	
		grant-S(A, T2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T2)
	read(B);	
	unlock(B);	
	display(A+B);	
lock-X(A);		
		grant-X(A, T2)
read(A);		
A = A + 50;		
write(A);		
unlock(A);		

Two Phase Locking Protocol

- This is a protocol which ensures serializability (conflict-serializable schedules).
- Two Phase Locking Protocol have two phases:
  - Phase 1: Growing Phase
    - transaction may obtain locks
    - transaction may not release locks
  - Phase 2: Shrinking Phase
    - transaction may release locks
    - transaction may not obtain locks

Example:

T1	
X (A)	
R (A)	
W(A)	**Growing Phase - Apply locks**
S (B)	
R (B)	
S (C)	
R (C)	
*	**Lock point**
U (A)	
U (B)	**Shrinking Phase - Release locks**
U(C)	

- The protocol ensures serializability.
  - It can be proved that the transactions can be serialized in the order of their lock points (i.e. the point where a transaction acquired its final lock or release first lock).

Example:

	T1	**T2**
Grant	X (A)	
	R (A)	
	W(A)	
		S (A) Wait
		R (A)
Grant	X (B)	
	R (B)	
	W(B)	
	U (A)	
	U (B)	

In the above example, T2 S(A) will not be granted until Transaction T1 performs U(A). Transaction T2 is waiting for Transaction T2 to release data item A. So the schedule is serializable and the execution order is T1→T2.

Example: **T1** **T2**

Grant lock-S(A)

lock-S(A) Grant

Grant lock-X(B)

Lock point of T1 *

Unlock(A)

lock-X(D) Grant

Unlock(B)

* Lock point of T2

Unlock(A)

Unlock(B)

In the above example, the schedule is serializable and the execution order is T1→T2 because of the order of lockpoints of transaction T1 and T2.

- Two-phase locking does not ensure freedom from deadlocks, irrecoverability and starvation.
- Cascading roll-back is possible under two-phase locking.

Example: **T1** **T2** **T3**

Grant X(A)

R(A)

W(A)

U(A)

S(A) Grant

R(A)

U(A)

S(A) Grant

R(A)

U(A)

Rollback

Abort

Fails *

Abort

- **Strict two-phase locking:**
  - Here a transaction must hold all its exclusive locks till it commits/ aborts.
  - A transaction may release all the shared locks after the lock point has been reached, but it cannot release any of the exclusive locks until the transaction commits or aborts.
  - This ensures that any data written by an uncommitted transaction is locked in exclusive mode until the transaction commits/ aborts and prevents other transactions from reading that data.
  - No cascading rollback possible.
  - This protocol solves dirty read problems so the schedule is always recoverable, cascadeless and strict schedule.

- **Rigorous two-phase locking:**
    - even stricter than strict 2PL protocol.
    - Here all locks (shared and exclusive) are held till commit/ abort.
    - A transaction is not allowed to release any lock until it commits/ aborts.
    - No cascading rollback (of course)
    - In this protocol transactions can be serialized in the order in which they commit.
    - Recovery is easy but concurrency is reduced.
    - Starvation and deadlock is still possible in this protocol.
- **Conservative two-phase locking:**
    - Before starting a transaction, acquire all the required locks.
    - This solves the deadlock problem.
    - Prior knowledge of all data items required at the beginning.

- There can be conflict serializable schedules that cannot be obtained if two-phase locking is used.
- However, in the absence of extra information (e.g., ordering of access to data), two-phase locking is needed for conflict serializability in the following sense:
    - Given a transaction Ti that does not follow two-phase locking, we can find a transaction Tj that uses two-phase locking, and a schedule for Ti and Tj that is not conflict serializable.

Timestamp Based Protocol

- Unique value is assigned to every transaction which tells the order when the transaction enters into the system.
- Each transaction is issued a timestamp when it enters the system. This protocol uses either system time or logical counter to be used as a time-stamp.
- Every transaction has a timestamp associated with it and the order is determined by the age of the transaction.
- If an old transaction Ti has time-stamp TS(Ti), a new transaction Tj is assigned a time-stamp TS(Tj) such that TS(Ti) < TS(Tj).
- The protocol ensures serializability among transactions in their conflicting read and write operations. This is the responsibility of the protocol system that the conflicting pairs of tasks should be executed according to the timestamp values of the transactions. It manages concurrent execution such that the time-stamps determine the serializability order. It gives priority to the older transactions- older transactions will complete first. So the execution order is older to younger (Ti→Tj).

- For example, a transaction created at 2:00 clock time would be older than all other transactions, which come after it. Any transaction Ty entering the system at 2:02 is two seconds younger and priory may be given to the older one.

Example: TS(T1)=10 TS(T2)=20 TS(T3)=30

T1	**T2**	**T3**
R(A)		
	R(A)	
		R(A)

So, **RTS(A)= 30**. T1 is an order transaction and T3 is a younger transaction. So, the execution order is older to younger (T1→T3).

Example: TS(T1)=10 TS(T2)=20 TS(T3)=30

T1	**T2**	**T3**
W(A)		
		W(A)
	W(A)	

So, **WTS(A)= 20**. T1 is an order transaction and T3 is a younger transaction. So, the execution order is older to younger (T1→T3).

- In order to assure such behavior, the protocol maintains for each data Q two timestamp values:
  - W-timestamp(Q) is the largest time-stamp of any transaction that executed write(Q) successfully. Timestamp of the last (Latest) transaction which performs write successfully on Q is called Write-TS(Q) or WTS(Q).
  - R-timestamp(Q) is the largest time-stamp of any transaction that executed read(Q) successfully. Timestamp of the last (Latest) transaction which performs read successfully on Q is called Read-TS(Q) or RTS(Q).
  - Every data item is given the latest read and write timestamp. This lets the system know when the last read and write operation was made on the data item.

- The timestamp ordering protocol ensures that any conflicting read and write operations are executed in timestamp order.

Rules

① Transaction T_i issues a Read(A) operation
   a) if $WTS(A) > TS(T_i)$, Rollback T_i
   b) Otherwise execute R(A) operation
      set $RTS(A) = Max\{RTS(A), TS(T_i)\}$

② Transaction T_i issues Write(A) operation
   a) if $RTS(A) > TS(T_i)$ then Rollback T_i
   b) if $WTS(A) > TS(T_i)$ then Rollback T_i
   c) otherwise execute Write(A) operation
      Set $WTS(A) = TS(T_i)$

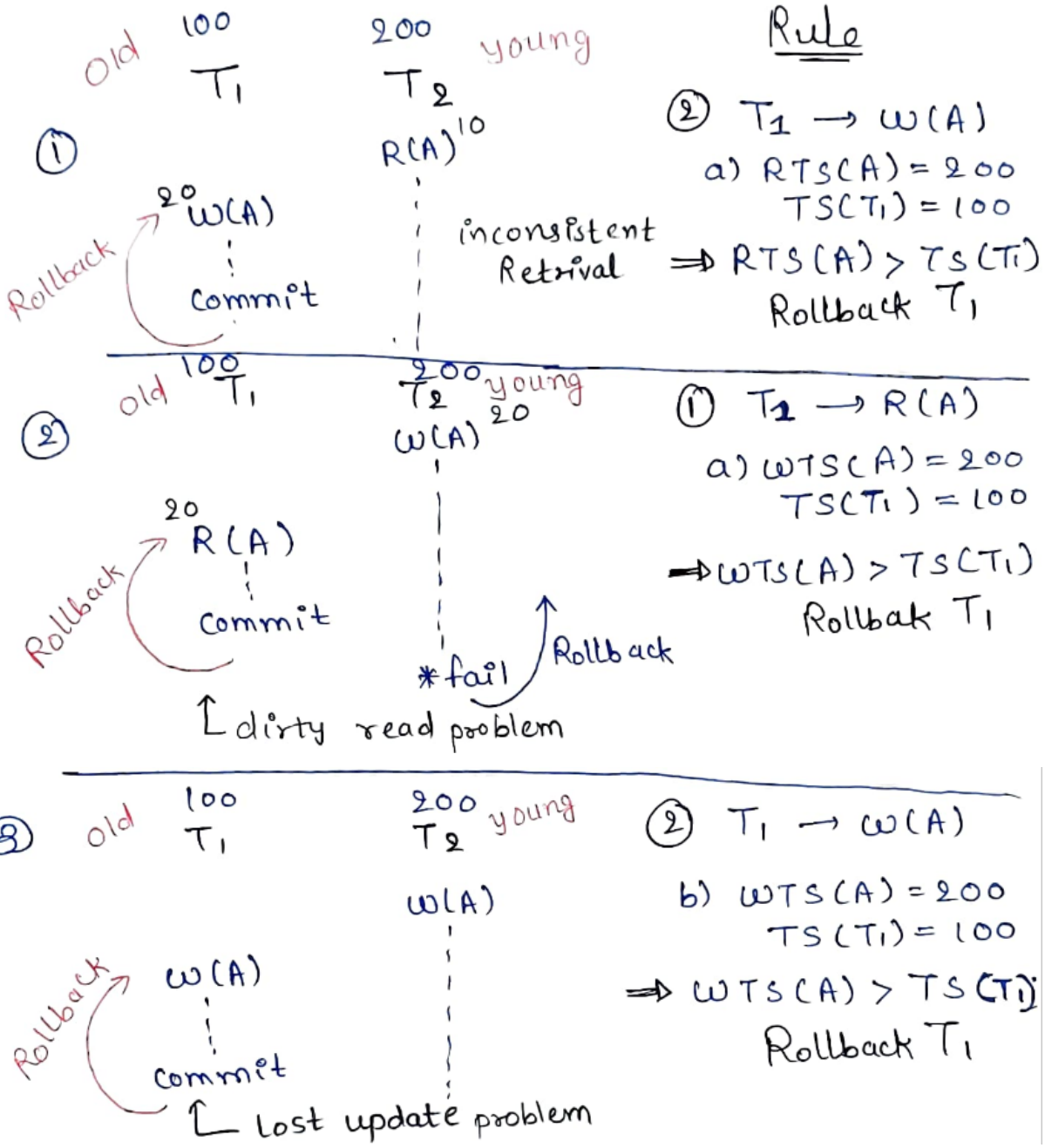
old
100 T_1 T_2 200 young

① R(A) ✓
   W(A)

② W(A) ✓
   R(A)

③ W(A) ✓
   W(A)

no problem ∵ T_1 older
completed 1st.


Old
100
T1
200
T2
young
R(A) 10
①
20
W(A)
Rollback
commit
inconsistent
Retrival
Rule
② T1 → W(A)
a) RTS(A) = 200
TS(T1) = 100
⇒ RTS(A) > TS(T1)
Rollback T1
Old
100
T1
200
T2
young
②
W(A) 20
① T2 → R(A)
a) WTS(A) = 200
TS(T1) = 100
⇒ WTS(A) > TS(T1)
Rollbak T1
20
R(A)
Rollback
commit
*fail
Rollback
dirty read problem
③
Old
100
T1
200
T2
young
W(A)
② T1 → W(A)
b) WTS(A) = 200
TS(T1) = 100
⇒ WTS(A) > TS(T1)
Rollback T1
W(A)
Rollback
commit
Lost update problem

Conflict / View Serializability.

+ → It ensures serializability. (older → yongest)
− → possibility of dirty read ✓, no restriction on C (so Irrecoverable schedules, cadscading rollback possible.)
+ → we allow or reject so no deadlock possible.
− → may suffer from Starvation & relatively slow, checking every op^ns condition for

Timestamp Ordering Protocol.

Eg:

100 oldest T_1	200 T_2	300 T_3 youngest
R(A)		
	R(B)	
W(C)		
		R(B)
R(C)		
	W(B)	
		W(A)

initial values

	A	B	C
	100	300	
RTS	∅	∅ ~~200~~	∅ 100
WTS	∅ 300	0	∅ 100

⇒ T_1 : R(A)

Rule ① a) if WTS(A) > TS(T_1) → false (∵ 0 ≯ 100)

WTS(A) = 0 TS(T_1) = 100

b) execute R(A)

RTS(A) = Max { RTS(A), TS(T_1) }

= Max { 0, 100 }

= 100

⇒ T_2 : R(B)

Rule ① a) if WTS(B) > TS(T_2) ⇒ 0 ≯ 200 ∴ false

WTS(B) = 0 TS(T_2) = 200

b) execute R(B)

RTS(B) = Max { RTS(B), TS(T_2) }

= Max { 0, 200 }

= 200

⇒ T_1 : W(C)

Rule ② a) if RTS(C) > TS(T_1) ⇒ 0 ≯ 100 ∴ false

RTS(C) = 0 TS(T_1) = 100

b) if WTS(C) > TS(T_1) ⇒ 0 ≯ 300 ∴ false

WTS(C) = 0 TS(T_1) = 100

c) execute Write(C)

WTS(C) = TS(T_1)

= 100

⇒ T_3 : R(B)

Rule ① a) WTS(B) > TS(T_3) ⇒ 0 ≯ 300 ∴ false

WTS(B) = 0 TS(T_3) = 300

b) execute R(B)

RTS(B) = Max { RTS(B), TS(T_3) }

= Max { 200, 300 }

= 300

⇒ T_1 : R(C)

Rule ① a) WTS(C) > TS(T_1) ⇒ 100 ≯ 100 ∴ false

WTS(C) = 100 TS(T_1) = 100

b) execute R(C)

RTS(C) = Max { RTS(C), TS(T_1) }

= Max { 0, 100 }

= 100

⇒ T_2 : W(B)

Rule (2) a) RTS(B) > TS(T_2) ⇒ 300 > 200

RTS(B) = 300 TS(T_2) = 200

Condition true ∴ Rollback T_2

200 300

old T_2 T_3 younger

⇒ T_2 will Rollback and now T_2 will start with new Timestamp that is greater than Timestamp of T_3.

⇒ T_3 : W(A)

Rule (2) a) RTS(A) > TS(T_3) ⇒ 100 ≯ 300 ∴ false

RTS(A) = 100 TS(T_3) = 300

b) WTS(A) > TS(T_3) ⇒ 0 ≯ 300 ∴ false

WTS(A) = 0 TS(T_3) = 300

c) execute W(A)

WTS(A) = TS(T_3)

= 300

T_1 & T_3 Successfully executed || T_2 Rollbacked.

⇒ Here T_2 rolledback. Because here youngest transaction T_3 has read the value of B that value is written/updated by older transaction T_2. and that is not allowed as R-W conflict occurs.

⇒ After following all read/write operations of T_1, T_2, T_3 transactions, we will get order of transaction that will be executed in below mentioned manner : $T_1 \rightarrow T_3 \rightarrow T_2$

* Thomas Write rule:

→ Modify time Stamping protocol to make some improvements in it.

→ May generates those schedules that are views. but not conflicts. and provides better concurrency

→ it modify time-stamping protcol in
  ↳ write operation
    ↳ when T_i request write(Q)
      if $WTS(Q) > TS(T_i)$

→ here T_i attempts to write value of Q.
  ↳ rather than rollbacking T_i → Timestamp
  ↳ write operation is ignored. → Thomas-rule

⇒ eg. Q = 10 14

TS → 5 10
T_i T_x

$W(Q)^{14}$

$^{12}W(Q)$ ← Blind write

$WTS(Q) > TS(T_i)$
$10 > 5$
∴ Rollback T_i

} Time Stamp

Q = 10 ~~12~~ 14

5 10
T_i T_x

$W(Q)^{12}$

$W(Q)^{14}$

Q is consistent
→ rather than rollback ignore W(Q).
→ Thomas write rule

---

eg: TS → 1 2 3
T_1 T_2 T_3

R(Q)

W(Q)

W(Q)

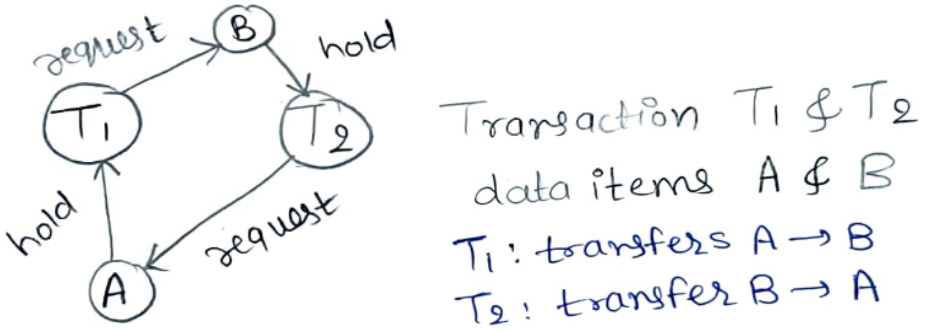
W(Q)

→ instead of rollback T_1 bcz of T_1 W(Q) we are ignoring T_1 W(Q) bez final value of Q is written by T_3 W(Q).

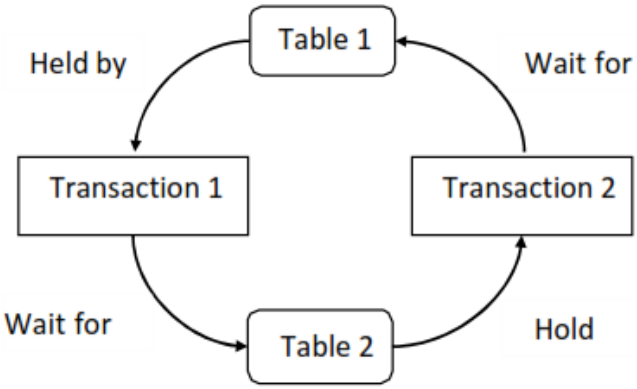
Deadlock

- A deadlock is a condition when two or more transactions are executing and each transaction is waiting for the other to finish but none of them are ever finished. So all the transactions will wait for infinite time and not a single transaction is completed.
- A deadlock is a condition where two or more transactions are waiting indefinitely for one another to give up lock, no operations ever get finished and are in waiting state forever.
- 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.

- **Example**:



- **Example**:



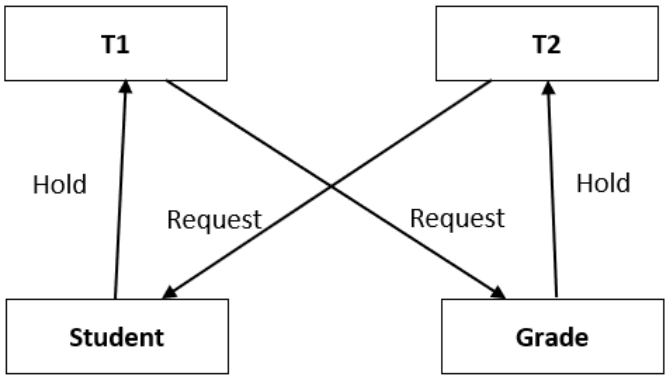
  - In the above figure there are two transactions 1 and 2 and two table's as table1 and table 2. Transaction 1 holds table 1 and waits for table 2. Transaction 2 holds table 2 and waits for table 1.
  - Now table 1 is wanted by transaction 2 and that is held by transaction 1 and the same way table 2 is wanted by transaction 1 and that is held by transaction 2. Until any one can't get this table they can't proceed further so this is called deadlock. Because both of these transactions have to wait for some resources.

- **Example**:

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

  - Since T3 is holding an exclusive mode lock on B and T4 is requesting a shared-mode lock on B, T4 is waiting for T3 to unlock B. Similarly, since T4 is holding a shared-mode lock on A and T3 is requesting an exclusive-mode lock on A, T3 is waiting for T4 to unlock A. Thus, we have arrived at a state where neither of these transactions can ever proceed with its normal execution. This situation is called deadlock.

- **Example:** In the student table, transaction T1 holds a lock on some rows and needs to update some rows in the grade table. Simultaneously, transaction T2 holds locks on some rows in the grade table and needs to update the rows in the Student table held by transaction T1.
  - Now transaction T1 is waiting for T2 to release its lock and similarly, transaction T2 is waiting for T1 to release its lock. All activities come to a halt state and remain at a standstill. It will remain in a standstill until the DBMS detects deadlock and aborts one the transactions.



- **When deadlock occurs**
  - A deadlock occurs when two separate transactions struggle for data items are held by one another.

- Deadlocks can occur in any concurrent system where transactions wait for each other and a cyclic chain can arise with each transaction waiting for the next one in the chain.
- Deadlock can occur in any system that satisfies the four conditions:

    1. **Mutual Exclusion Condition**: All the data items involved in the schedule will be used by transactions in a mutually exclusive way i.e. one by one. Only one transaction at a time can use a data item or each data item assigned to one transaction only at that time or that data item is available to use.
    2. **Hold and Wait Condition**: Transactions already holding data items may request new data items.
    3. **No Preemption Condition**: Only a transaction holding a data item can release it voluntarily after that transaction has completed its task or previously granted data items cannot forcibly taken away from any transaction.
    4. **Circular Wait Condition**: two or more transactions form a circular chain where each transaction requests a data item that the next transaction in the chain holds.

Deadlock Avoidance

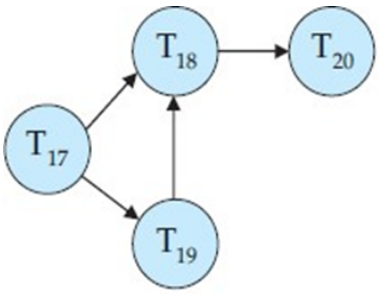
- When a database is stuck in a deadlock state then it is better to avoid the database rather than aborting or restarting the database. This is a waste of time and resources.
- Deadlock avoidance mechanism is used to detect any deadlock situation in advance. System maintains a set of data using which it takes a decision to entertain a new request or not to be in a safe state (deadlock will not occur).
- A method like “wait for graph” is used for detecting the deadlock situation but this method is suitable only for the smaller database. For the larger database, a deadlock prevention method can be used.

Deadlock Detection and Recovery

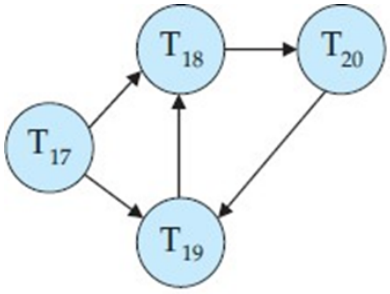
Deadlock Detection

- In a database, when a transaction waits indefinitely to obtain a ock, then the DBMS should detect whether the transaction is involved in a deadlock or not. The lock manager maintains a Wait for graph to detect the deadlock cycle in the database. A graph is created based on the transaction and their lock. If the graph has a cycle or closed loop then there is a deadlock.
- Deadlocks can be described precisely in terms of a directed graph called a wait for graph. This graph consists of a pair G = (V, E), where V is a set of vertices and E is a set of edges. The set of vertices consists of all the transactions in the system.

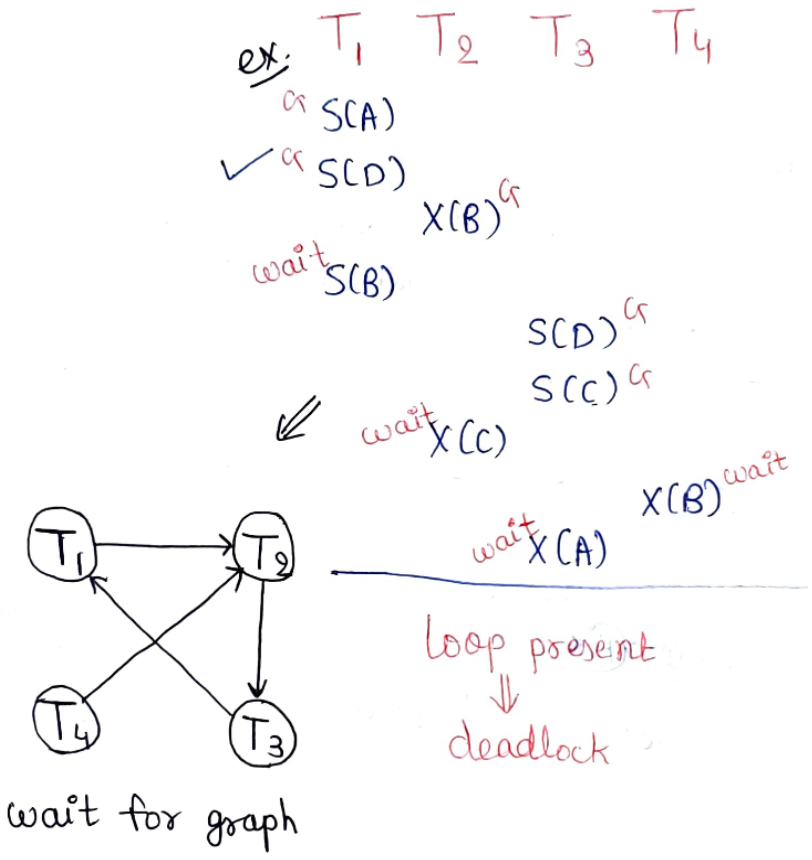
- Each element in the set E of edges is an ordered pair Ti → Tj. If Ti → Tj is in E, then there is a directed edge from transaction Ti to Tj , implying that transaction Ti is waiting for transaction Tj to release a data item that it needs.
- When transaction Ti requests a data item currently being held by transaction Tj , then the edge Ti → Tj is inserted in the wait-for graph. This edge is removed only when transaction Tj is no longer holding a data item needed by transaction Ti .
- A deadlock exists in the system if and only if the wait-for graph contains a cycle. Each transaction involved in the cycle is said to be deadlocked. To detect deadlocks, the system needs to maintain the wait-for graph, and periodically to invoke an algorithm that searches for a cycle in the graph.

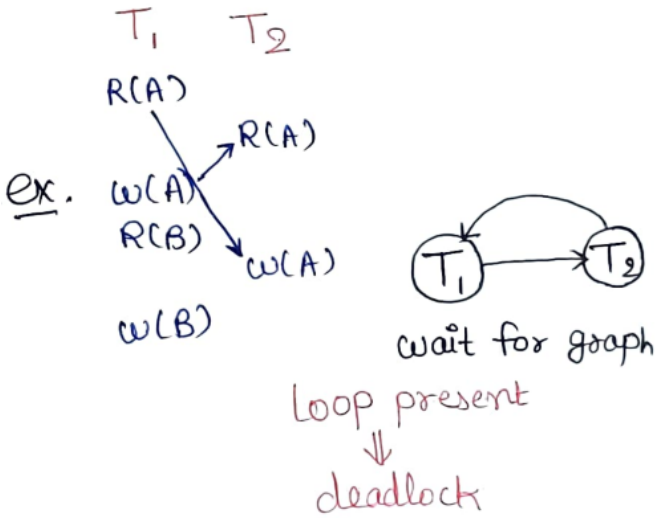


Wait-for graph with no cycle.



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. The most common solution is to roll back one or more transactions to break the deadlock. Three actions need to be taken:

1. **Selection of a victim**
   - Given a set of deadlocked transactions, we must determine which transaction (or transactions) to roll back to break the deadlock. We should roll back those transactions that will incur the minimum cost. Unfortunately, the term minimum cost is not a precise one. Many factors may determine the cost of a rollback, including:
     a. How long the transaction has been computed, and how much longer the transaction will compute before it completes its designated task.
     b. How many data items the transaction has used, and how many more data items the transaction needs for it to complete.
     c. How many transactions will be involved in the rollback.
2. **Rollback**
   - Once we have decided that a particular transaction must be rolled back, we must determine how far this transaction should be rolled back. The simplest solution is a total rollback: Abort the transaction and then restart it. However, it is more effective to roll back the transaction only as far as necessary to break the deadlock. This is called a partial rollback.
3. **Starvation**
   - In a system where the selection of victims is based primarily on cost factors, it may happen that the same transaction is always picked as a victim. As a result, this transaction never completes its designated task, thus there is starvation. We must ensure that a 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.

Deadlock Prevention

- Deadlock prevention method is suitable for a large database.
- If the resources are allocated in such a way that deadlock never occurs, then the deadlock can be prevented.
- The Database Management system analyzes the operations of the transaction whether they can create a deadlock situation or not. If they do, then the DBMS never allows that transaction to be executed.
- A protocol ensures that the system will never enter into a deadlock state.

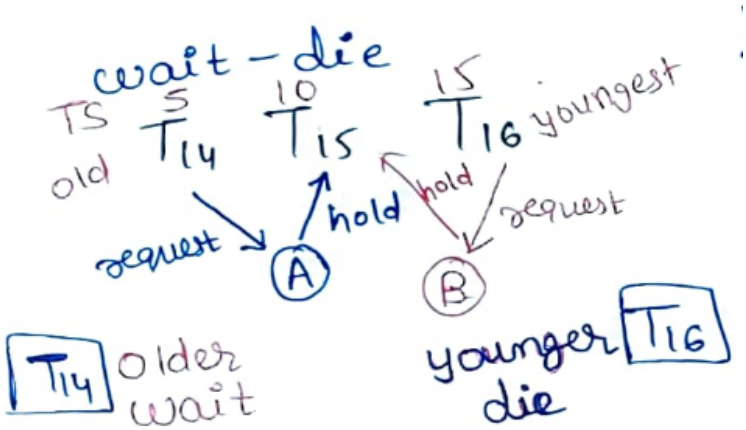
- Some prevention strategies :
    - Require that each transaction locks all its data items before it begins execution (predeclaration). Moreover, either all are locked in one step or none are locked.
    - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial.
    - There are two main disadvantages to this protocol:
        - it is often hard to predict, before the transaction begins, what data items need to be locked;
        - data-item utilization may be very low, since many of the data items may be locked but unused for a long time.

- **Based on timestamps**:

    - In this scheme, if a transaction requests for a resource/ data item which is already held with a conflicting lock by another transaction then the DBMS simply checks the timestamp of both transactions.

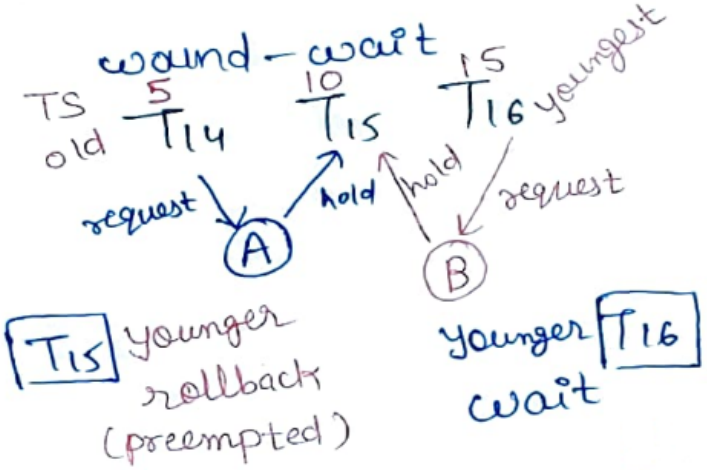
    1. **Wait-die scheme**

        - Wait-die scheme is a non- preemptive technique.
        - If an older transaction is requesting a resource which is held by a younger transaction, then the older transaction is allowed to wait for it till it is available.
        - If a younger transaction is requesting a resource which is held by an older transaction, then the younger transaction is killed.
        - When a 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 T14, T15, and T16 have timestamps 5, 10, and 15, respectively. If T14 requests a data item held by T15, then T14 will wait. If T16 requests a data item held by T15, then T16 will be rolled back.



2. Wound-wait scheme

- The wound–wait scheme is a preemptive technique.
- If an older transaction is requesting a resource which is held by a younger transaction, then the older transaction forces the younger transaction to kill the transaction and release the resource.
- If a younger transaction is requesting a resource which is held by an older transaction, then the younger transaction is allowed to wait till the older transaction will release it.
- When a transaction Ti requests a data item currently held by Tj , Ti is allowed to wait only if it has a timestamp larger than that of Tj (that is, Ti is younger than Tj). Otherwise, Tj is rolled back (Tj is wounded by Ti).
- For example, with transactions T14, T15, and T16, if T14 requests a data item held by T15, then the data item will be preempted from T15, and T15 will be rolled back. If T16 requests a data item held by T15, then T16 will wait.



- The major problem with both of these schemes is that unnecessary rollbacks may occur.

- **Timeout-Based Schemes** :
    - 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 time out, and it rolls itself back and restarts. If there was in fact a deadlock, one or more transactions involved in the deadlock will time out and roll back, allowing the others to proceed. So deadlocks never occur.
    - Simple to implement; but difficult to determine the good value of the timeout interval. Starvation is possible.