Concurrency Controls Techniques

Concurrency Control

- Concurrency control is the procedure in DBMS for managing simultaneous operations without conflicting with each another. Concurrent access is quite easy if all users are just reading data. There is no way they can interfere with one another. Though for any practical database, would have a mix of reading and WRITE operations and hence the concurrency is a challenge.
- Concurrency control is used to address such conflicts which mostly occur with a multi-user system. It helps you to make sure that database transactions are performed concurrently without violating the data integrity of respective databases.
- Therefore, concurrency control is a most important element for the proper functioning of a system where two or multiple database transactions that require access to the same data, are executed simultaneously.

Potential problems of Concurrency

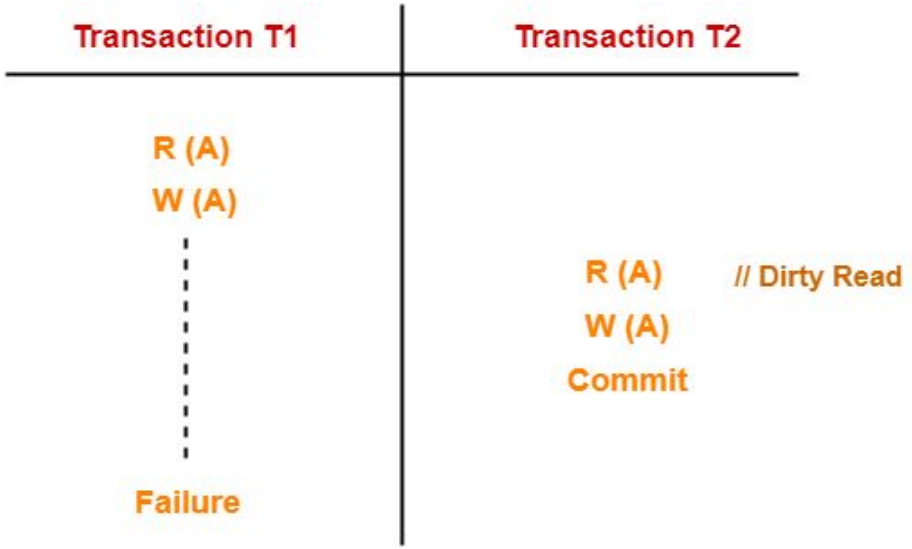
- When multiple transactions execute concurrently in an uncontrolled or unrestricted manner, then it might lead to several problems.
- Such problems are called as **concurrency problems**.

The concurrency problems are:-

Dirty Read Problem

This read is called as dirty read because-

- There is always a chance that the uncommitted transaction might roll back later.
- Thus, uncommitted transaction might make other transactions read a value that does not even exist.
- This leads to inconsistency of the database.
- Dirty read does not lead to inconsistency always.
- It becomes problematic only when the uncommitted transaction fails and roll backs later due to some reason.



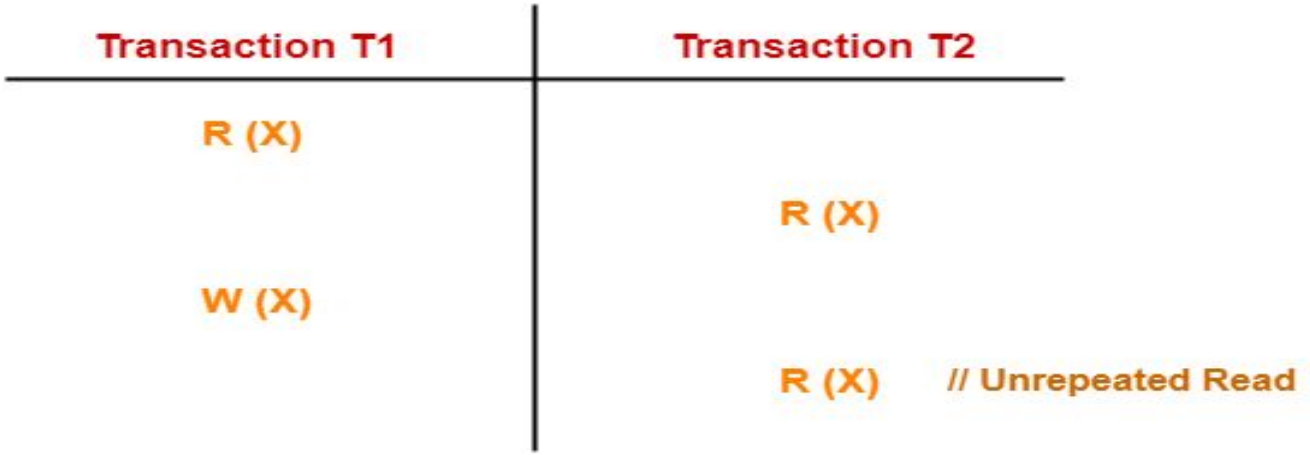
- T1 reads the value of A.
- T1 updates the value of A in the buffer.
- T2 reads the value of A from the buffer.
- T2 writes the updated the value of A.
- T2 commits.
- T1 fails in later stages and rolls back.

In this example,

- T2 reads the dirty value of A written by the uncommitted transaction T1.
- T1 fails in later stages and roll backs.
- Thus, the value that T2 read now stands to be incorrect.
- Therefore, database becomes inconsistent.

Unrepeatable Read Problem

This problem occurs when a transaction gets to read unrepeated i.e. different values of the same variable in its different read operations even when it has not updated its value.



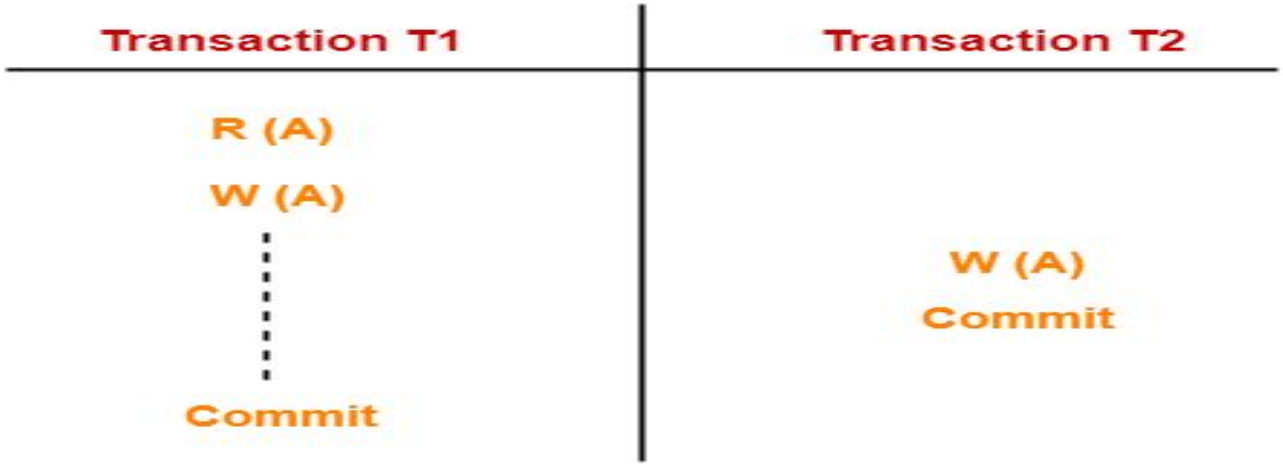
- T1 reads the value of X (= 10 say).
- T2 reads the value of X (= 10).
- T1 updates the value of X (from 10 to 15 say) in the buffer.
- T2 again reads the value of X (but = 15).

In this example,

- T2 gets to read a different value of X in its second reading.
- T2 wonders how the value of X got changed

Lost Update Problem

This problem occurs when multiple transactions execute concurrently and updates from one or more transactions get lost.



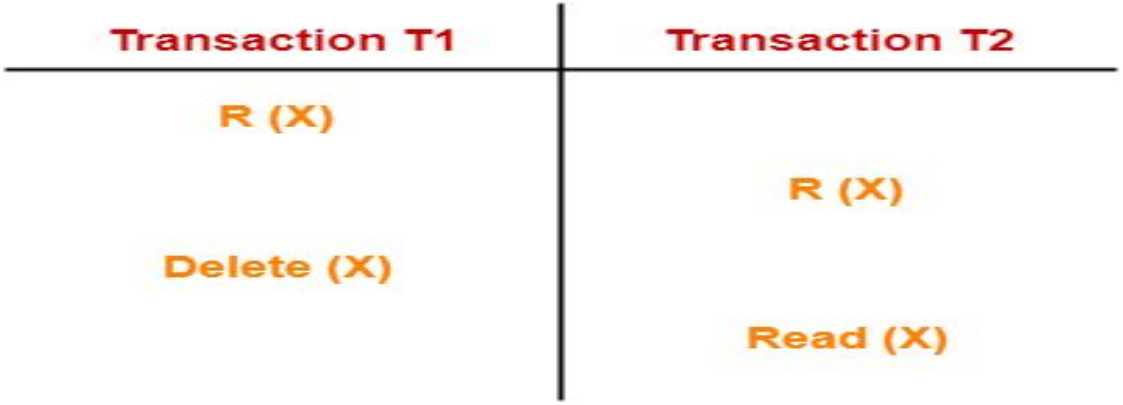
- T1 reads the value of A (= 10 say).
- T1 updates the value to A (= 15 say) in the buffer.
- T2 does blind write A = 25 (write without read) in the buffer.
- T2 commits.
- When T1 commits, it writes A = 25 in the database.

In this example,

- T1 writes the over written value of X in the database.
- Thus, update from T1 gets lost.

Phantom Read Problem

This problem occurs when a transaction reads some variable from the buffer and when it reads the same variable later, it finds that the variable does not exist.



- T1 reads X.
- T2 reads X.
- T1 deletes X.
- T2 tries reading X but does not find it.

In this example,

- T2 finds that there does not exist any variable X when it tries reading X again.
- T2 wonders who deleted the variable X

Reasons for using Concurrency control method is DBMS:

- To apply Isolation through mutual exclusion between conflicting transactions
- To resolve read-write and write-write conflict issues
- To preserve database consistency through constantly preserving execution obstructions
- The system needs to control the interaction among the concurrent transactions. This control is achieved using concurrent-control schemes.
- Concurrency control helps to ensure serializability

Concurrency Control Protocols

- Lock-Based Protocols
- Two Phase
- Timestamp-Based Protocols
- Validation-Based Protocols

Lock Based Protocol

- Lock is a mechanism to control concurrent access to data item
- Data items can be locked in two modes:
  - **Exclusive (X) Mode** :- Data item can be both read as well as written. X-lock is requested using lock-X instruction
  - **Shared (S) Mode** :- Data item can only be read. S-lock is requested using lock-S instruction
- Lock requests are made to concurrency-control manager
- Transaction can proceed only after request is granted

T1	T2
S-Lock(A) R(A) Unlock(A)	Lock-X(A) R(A) W(A) Unlock(A)

Lock-compatibility Matrix :

Grant \ Request	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 transaction
- 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

Problem With Lock-Based Protocols

- May not sufficient to produce only serializable schedule

T1	T2
X(A) R(A) W(A) U(A) X(B) R(B) W(B) U(B)	 S(A) R(A) U(A)

- May not free from irrecoverability

T1	T2
R(A) W(A) Fail	 R(A) Commit

Problem With Lock-Based Protocols

- May not free from deadlock

T1	T2
G X(A) W X(B)	 G X(B) W X(A)

- May not free from starvation

T1	T2	T3	T4
  W X(A) . . . . . G	S(A) . . . U(A)	 S(A) . . . U(A)	 S(A) U(A)

Two Phase Locking (2PL) Protocol

Two-Phase locking protocol which is also known as a 2PL protocol. It is also called P2L. In this type of locking protocol, the transaction should acquire a lock after it releases one of its locks.

The Two-Phase Locking protocol allows each transaction to make a lock or unlock request in two steps:

Growing Phase: Locks are acquired and no locks are released .

Shrinking Phase: Locks are released no locks are acquired

T1	T2
X(A) R(A) W(A) S(B) R(B) U (A) U(B)	 S(A) R(A)

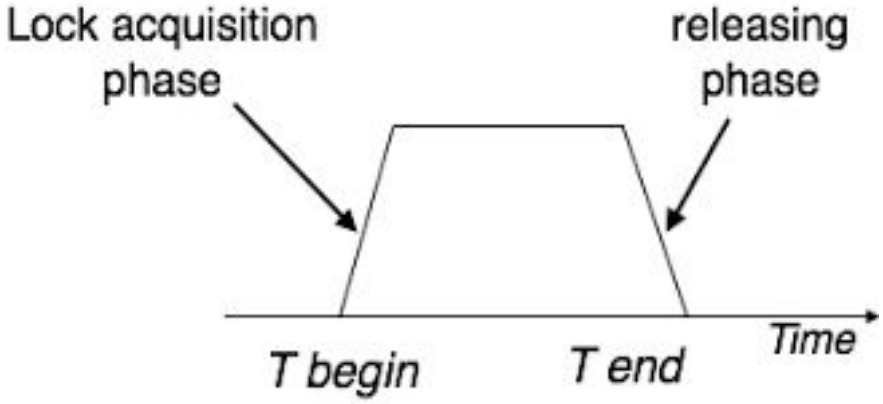
Growing

Shrinking

T1	T2
LOCK- S(A) G LOCK -X(B) Unlock(A) Unlock(B)	 LOCK- S(A) G LOCK-X(D) Unlock (A) S Unlock(D)

S

Two Phase Locking (2PL) Protocol



Advantage:

- Always ensure serializability

Disadvantages:

- May not free from irrecoverability
- May not free from cascading rollback
- May not free from deadlock
- May not free from starvation

Categories of Two Phase Locking

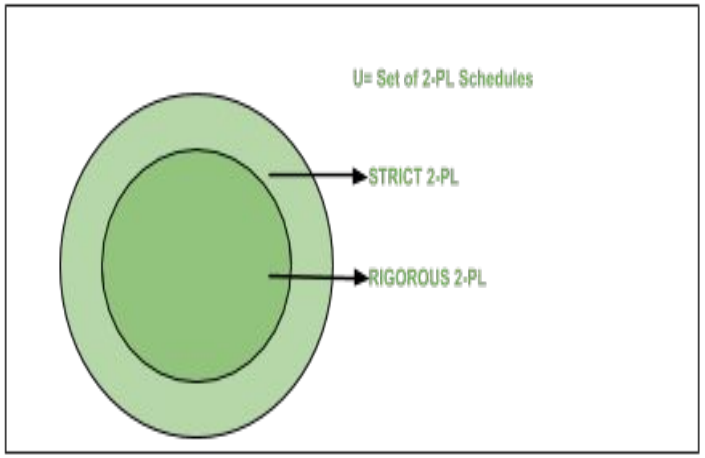
Strict 2PL : It should satisfy the basic 2pl and all exclusive lock should hold until commit /abort

T1	T2
X(A) R(A) W(A) . . . . Commit U(A)	 S(A) R(A)

Rigorous 2PL : It should satisfy the basic 2pl and all shared , exclusive lock should hold until commit /abort.

T1	T2
X(A) R(A) W(A) . . . . Commit U(A)	 S(A) R(A) Commit U(A)

Conservative 2PL : A.K.A **Static 2-PL**, this protocol requires the transaction to lock all the items it access before the Transaction begins execution by predeclaring its read-set and write-set. If any of the predeclared items needed cannot be locked, the transaction does not lock any of the items, instead, it waits until all the items are available for locking. So the operation on data cannot start until we lock all the items required.



Timestamp Ordering Protocol

- The Timestamp Ordering Protocol is used to order the transactions based on their Timestamps. The order of transaction is nothing but the ascending order of the transaction creation.
- The priority of the older transaction is higher that's why it executes first. To determine the timestamp of the transaction, this protocol uses system time or logical counter.
- The lock-based protocol is used to manage the order between conflicting pairs among transactions at the execution time. But Timestamp based protocols start working as soon as a transaction is created.
- Let's assume there are two transactions T1 and T2. Suppose the transaction T1 has entered the system at 007 times and transaction T2 has entered the system at 009 times. T1 has the higher priority, so it executes first as it is entered the system first.
- The timestamp ordering protocol also maintains the timestamp of last 'read' and 'write' operation on a data.

T1	T2
Time stamp(007)	Time stmp(009)

TS(T_I) denotes the timestamp of the transaction Ti.
R_TS(X) denotes the last (latest) Read time-stamp of transaction).
W_TS(X) denotes the last (latest) Write time-stamp of transaction.

T1(007) Older	T2(009) younger

TS(T1)=007
TS(T2)=009

T1(10)	T2 (20)	T3(30)
R(A)	R(A)	R(A)

R_TS(A)=30

T1 (100)	T2 (200)	T3 (300)
W(A)	W(A)	W(A)

W_TS(A)=200

Basic Timestamp ordering protocol works as follows:

1. Check the following condition whenever a transaction Ti issues a **Read (X)** operation:

- If W_TS(X) >TS(Ti) then the operation is rejected.
- If W_TS(X) <= TS(Ti) then the operation is executed.

2. Check the following condition whenever a transaction Ti issues a **Write(X)** operation:

- If TS(Ti) < R_TS(X) then the operation is rejected.
- If TS(Ti) < W_TS(X) then the operation is rejected and Ti is rolled back otherwise the operation is executed.

T1(10)	T2(20)
R(A)	
	W(A)

T1(10)	T2(20)
W(A)	
	R(A)

T1(10)	T2(20)
W(A)	
	W(A)

T1(10)	T2(20)
	R(A)
W(A)	

Write operation request.
Operation rejected

T1(10)	T2(20)
R(A)	
	W(A)

Read operation request.
Operation rejected

Advantages and Disadvantages of TO protocol:

- TO protocol ensures serializability since the precedence graph is as follows:

Image: Precedence Graph for TS ordering

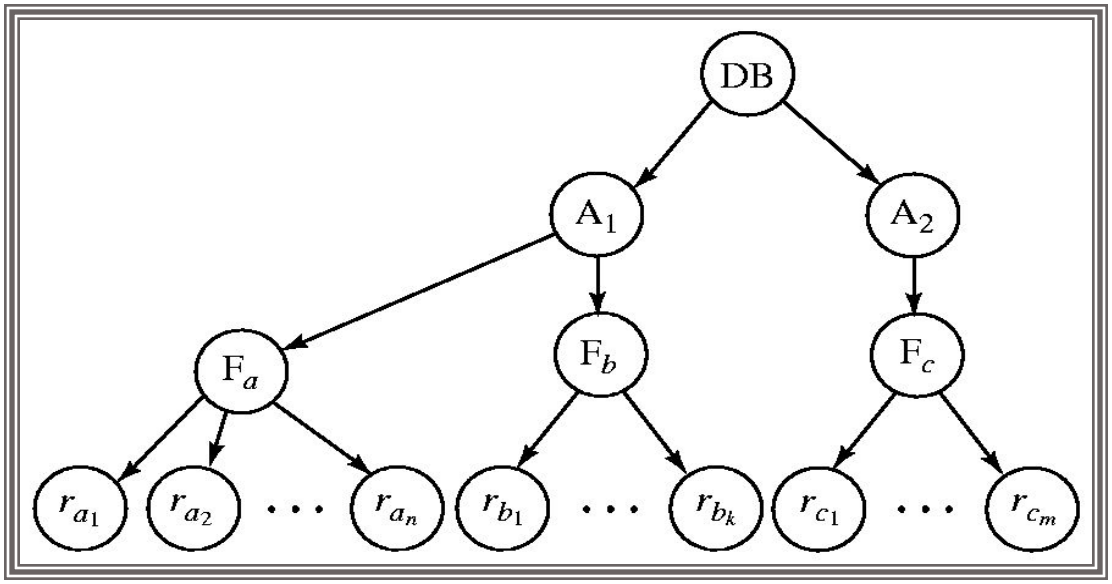
- TS protocol ensures freedom from deadlock that means no transaction ever waits.
- But the schedule may not be recoverable and may not even be cascade- free.

Thomas' Write Rule
This rule states if TS(Ti) < W-timestamp(X), then the operation is rejected and T_i is rolled back.

Multiple Granularity

- Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones
- Can be represented graphically as a tree (but don't confuse with tree locking protocol)
- When a transaction locks a node in the tree *explicitly, it implicitly locks* all the node's descendents in the same mode.
- Granularity of locking (level in tree where locking is done):
  - **Fine granularity (lower in tree): high concurrency, high locking** Overhead
  - **Coarse granularity (higher in tree): low locking overhead, low** concurrency

Example of Granularity Hierarchy



The levels, starting from the coarsest (top) level are

- Database
- Area
- File
- Record

Intention Lock Modes

In addition to S and X lock modes, there are three additional lock modes with multiple granularity:

intention-shared (IS): indicates explicit locking at a lower level of the tree but only with shared locks.

intention-exclusive (IX): indicates explicit locking at a lower level with exclusive or shared locks

shared and intention-exclusive (SIX): the sub tree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.

intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.

Compatibility Matrix with Intention Lock Modes

The compatibility matrix for all lock modes is:

	IS	IX	S	S IX	X
IS	✔	✔	✔	✔	×
IX	✔	✔	×	×	×
S	✔	×	✔	×	×
S IX	✔	×	×	×	×
X	×	×	×	×	×

Multiple Granularity Locking Scheme

Transaction T_i can lock a node Q, using the following rules:

1. The lock compatibility matrix must be observed.
2. The root of the tree must be locked first, and may be locked in any mode.
3. A node Q can be locked by T_i in S or IS mode only if the parent of Q is currently locked by T_i in either IX or IS mode.
4. A node Q can be locked by T_i in X, SIX, or IX mode only if the parent of Q is currently locked by T_i in either IX or SIX mode.
5. T_i can lock a node only if it has not previously unlocked any node (that is, T_i is two-phase).
6. T_i can unlock a node Q only if none of the children of Q are currently locked by T_i.

Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.

Validation-Based Protocol

In the validation based protocol, the transaction is executed in the following three phases:

Read phase: In this phase, the transaction T is read and executed. It is used to read the value of various data items and stores them in temporary local variables. It can perform all the write operations on temporary variables without an update to the actual database.

Validation phase: In this phase, the temporary variable value will be validated against the actual data to see if it violates the serializability.

Write phase: If the validation of the transaction is validated, then the temporary results are written to the database or system otherwise the transaction is rolled back.

Also called as **optimistic concurrency control** since transaction executes fully in the hope that all will go well during validation

Here each phase has the following different timestamps:

Start(Ti): It contains the time when Ti started its execution.

Validation (T_i): It contains the time when Ti finishes its read phase and starts its validation phase.

Finish(Ti): It contains the time when Ti finishes its write phase.

Multiversion Schemes

- Multiversion schemes keep old versions of data item to increase concurrency.
  - Multiversion Timestamp Ordering
  - Multiversion Two-Phase Locking
- Each successful **write** results in the creation of a new version of the data item written.
- Use timestamps to label versions.
- When a **read**(Q) operation is issued, select an appropriate version of Q based on the timestamp of the transaction, and return the value of the selected version.
- **read**s never have to wait as an appropriate version is returned immediately.

Multiversion Timestamp Ordering

Each data item Q has a sequence of versions $<Q_1, Q_2, ..., Q_m>$. Each version Q_k contains three data fields:

Content -- the value of version Q_k.

W-timestamp(Q_k) -- timestamp of the transaction that created (wrote) version Q_k

R-timestamp(Q_k) -- largest timestamp of a transaction that successfully read version Q_k

when a transaction T_i creates a new version Q_k of Q, Q_k's W-timestamp and R-timestamp are initialized to TS(T_i).

R-timestamp of Q_k is updated whenever a transaction T_j reads Q_k, and TS(T_j) > R-timestamp(Q_k).

Snapshot Isolation concurrency control

- A transactions reads a snapshot of data (Committed version only)
- Before committing , a transactions checks if any concurrent transaction wrote the same data.
- Readers do not block a writers and writers do not block a readers.
- Only checks for write-write conflicts
  - Write set of concurrent transaction must be disjoint.

Deadlock in DBMS

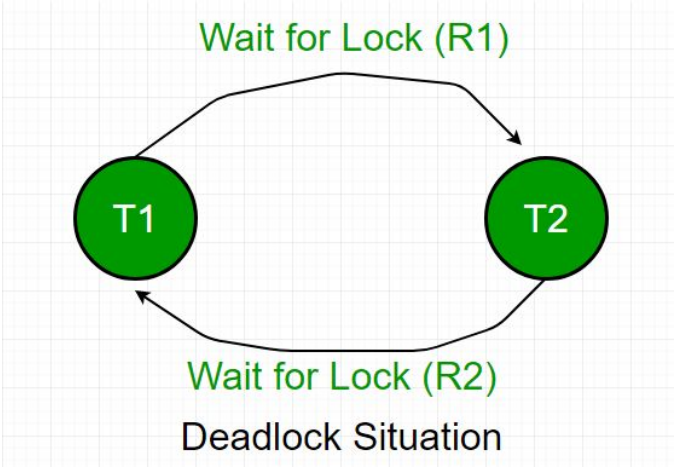
In a database, a deadlock is an unwanted situation in which two or more transactions are waiting indefinitely for one another to give up locks. Deadlock is said to be one of the most feared complications in DBMS as it brings the whole system to a Halt.

Deadlock Detection –

When a transaction waits indefinitely to obtain a lock, The database management system should detect whether the transaction is involved in a deadlock or not.

Wait-for-graph is one of the methods for detecting the deadlock situation. This method is suitable for smaller database. In this method a graph is drawn based on the transaction and their lock on the resource. If the graph created has a closed loop or a cycle, then there is a deadlock.

For the above mentioned scenario the Wait-For graph is drawn below



Deadlock prevention –

For large database, deadlock prevention method is suitable. A deadlock can be prevented if the resources are allocated in such a way that deadlock never occur. The DBMS analyzes the operations whether they can create deadlock situation or not, If they do, that transaction is never allowed to be executed.

Deadlock prevention mechanism proposes two schemes :

wait-die scheme

- 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

wound-wait scheme

- Older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.
- May be fewer rollbacks than *wait-die* scheme.

Thank You