

Concurrency Control

Concurrency Control is the working concept that is required for controlling and managing the concurrent execution of database operations and thus avoiding the inconsistencies in the database. Thus, for maintaining the concurrency of the database, we have the concurrency control protocols.

Concurrency Control Protocols

The concurrency control protocols ensure the *atomicity*, *consistency*, *isolation*, *durability* and *serializability* of the concurrent execution of the database transactions. Therefore, these protocols are categorized as:

- Lock Based Concurrency Control Protocol
- Time Stamp Concurrency Control Protocol
- Validation Based Concurrency Control Protocol

Lock-Based Protocol

In this type of protocol, any transaction cannot read or write data until it acquires an appropriate lock on it. There are two types of lock:

1. Shared lock(S) :

- It is also known as a Read-only lock. In a shared lock, the data item can only read by the transaction.
- It can be shared between the transactions because when the transaction holds a lock, then it can't update the data on the data item.

2. Exclusive lock (X):

- In the exclusive lock, the data item can be both reads as well as written by the transaction.
- This lock is exclusive, and in this lock, multiple transactions do not modify the same data simultaneously.

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.

There are four types of lock protocols available:

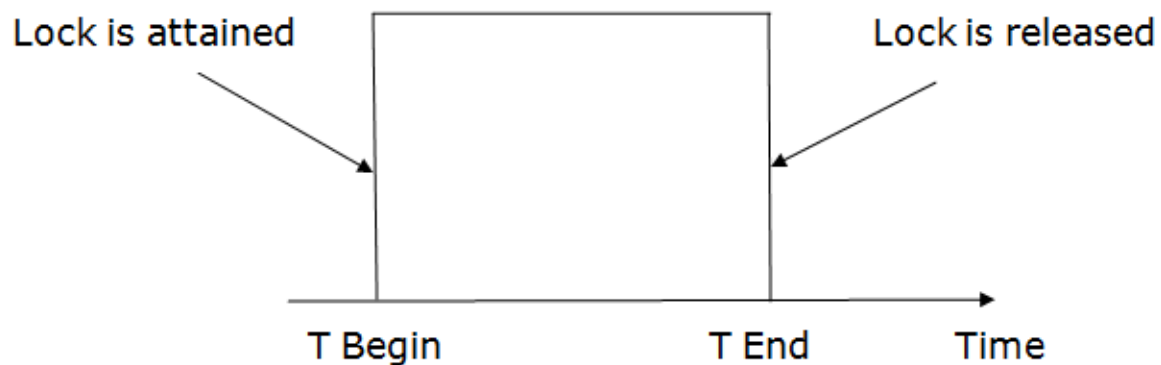
1. Simplistic lock protocol

It is the simplest way of locking the data while transaction. Simplistic lock-based protocols **allow all the transactions to get the lock on the data before insert or delete or update on it**. It will unlock the data item after completing the transaction.

Problem of conflicts and inconsistent database state

2. Pre-claiming Lock Protocol

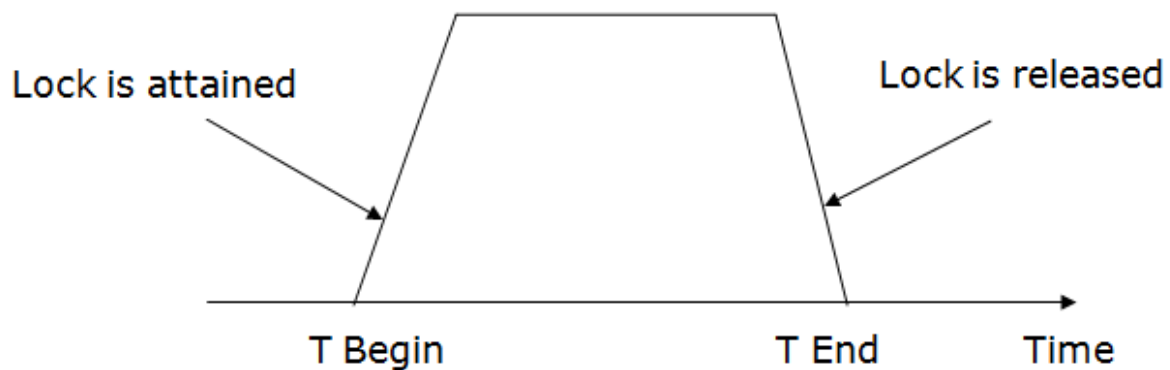
- Pre-claiming Lock Protocols evaluate the transaction to list all the data items on which they need locks.
- **Before initiating an execution of the transaction, it requests DBMS for all the lock on all those data items.**
- If all the locks are granted then this protocol allows the transaction to begin. When the transaction is completed then it releases all the lock.
- **If all the locks are not granted then this protocol allows the transaction to rolls back and waits until all the locks are granted.**



3. Two-phase locking (2PL)

- The two-phase locking protocol divides the execution phase of the transaction into three parts.

- In the **first phase**, when the execution of the transaction starts, it seeks permission for the locks it requires and it acquires all the locks.
- The **second phase** is started as soon as the transaction releases its first lock and further the transaction cannot demand any new locks. It only releases the acquired locks.



There are two phases of 2PL:

Growing phase: In the growing phase, a new lock on the data item may be acquired by the transaction, but none can be released.

Shrinking phase: In the shrinking phase, existing lock held by the transaction may be released, but no new locks can be acquired.

In the below example, if lock conversion is allowed then the following phase can happen:

1. Upgrading of lock (from S(a) to X (a)) is allowed in growing phase.
2. Downgrading of lock (from X(a) to S(a)) must be done in shrinking phase.

Example:

	T1	T2
0	LOCK-S(A)	
1		LOCK-S(A)
2	LOCK-X(B)	
3	—	—
4	UNLOCK(A)	
5		LOCK-X(C)
6	UNLOCK(B)	
7		UNLOCK(A)
8		UNLOCK(C)
9	—	—

The following way shows how unlocking and locking work with 2-PL.

Transaction T1:

- **Growing phase:** from step 1-3
- **Shrinking phase:** from step 5-7

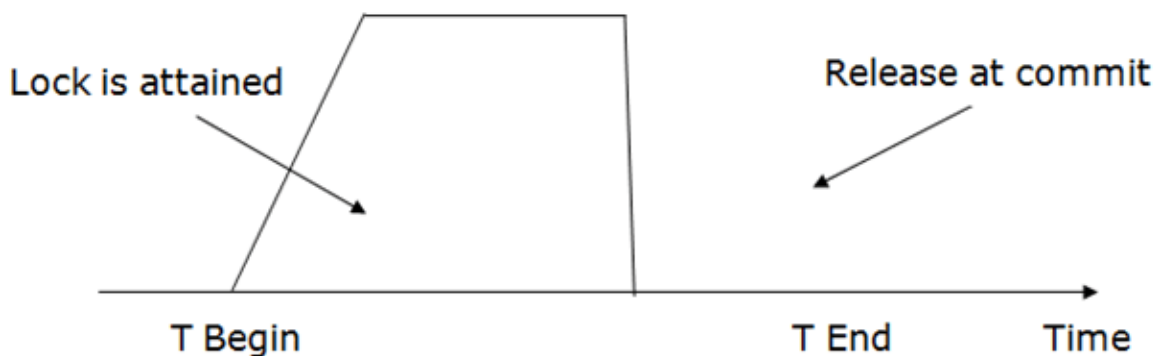
- **Lock point:** at 3

Transaction T2:

- **Growing phase:** from step 2-6
- **Shrinking phase:** from step 8-9
- **Lock point:** at 6

4. Strict Two-phase locking (Strict-2PL)

- The first phase of Strict-2PL is similar to 2PL. In the first phase, after acquiring all the locks, the transaction continues to execute normally.
- The only difference between 2PL and strict 2PL is that Strict-2PL **does not release a lock after using it.**
- Strict-2PL waits until the whole transaction to **commit, and then it releases all the locks at a time.**
- Strict-2PL protocol does not have shrinking phase of lock release.



It does not have cascading abort as 2PL does.

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.

Basic Timestamp ordering protocol works as follows:

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

- If $W_TS(X) > TS(T_i)$ then the operation is rejected.(this is older transaction that has requested read(X) before other transaction has written(X)
- If $W_TS(X) \leq TS(T_i)$ then the operation is executed.

- Timestamps of all the data items are updated.
2. Check the following condition whenever a transaction T_i issues a **Write(X)** operation:

- If $TS(T_i) < R_TS(X)$ then the operation is rejected. (T_i is older transaction but is issuing write(X) after other transaction has already read(X))
- If $TS(T_i) < W_TS(X)$ then the operation is rejected and T_i is rolled back otherwise the operation is executed. (T_i is older transaction but is issuing write(X) after other transaction has already write(X))

Where,

TS(T_i) denotes the timestamp of the transaction T_i .

R_TS(X) denotes the Read time-stamp of data-item X.

W_TS(X) denotes the Write time-stamp of data-item X.

Advantages and Disadvantages of TO protocol:

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

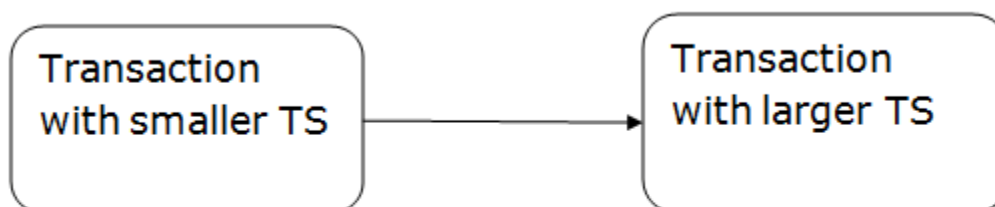


Image: Precedence Graph for TS ordering

- 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.**

(as multiple roll backs for rejected transactions are required)

Validation Based Protocol

Validation phase is also known as optimistic concurrency control technique. In the validation based protocol, the transaction is executed in the following three phases:

1. **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.
2. **Validation phase:** In this phase, the temporary variable value will be validated against the actual data to see if it violates the serializability.
3. **Write phase:** If the validation of the transaction is successful, then the temporary results are written to the database or system otherwise the transaction is rolled back.

Here each phase has the following different timestamps:

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

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

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

- **This protocol is used to determine the time stamp for the transaction for serialization using the time stamp of the validation phase**, as it is the actual phase which determines if the transaction will commit or rollback.
- Hence $TS(T) = \text{validation}(T)$.
- **The serializability is determined during the validation process.** It can't be decided in advance.
- While executing the transaction, it ensures a greater degree of concurrency and also less number of conflicts.
- Thus it contains transactions which have less number of rollbacks.

Deadlock in DBMS

A deadlock is a condition where 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 no task ever gets finished and is in waiting state forever.

For example: In the student table, transaction T_1 holds a lock on some rows and needs to update some rows in the grade table. Simultaneously, transaction T_2 holds locks on some rows in the grade table and needs to update the rows in the Student table held by Transaction T_1 .

Now, the main problem arises. 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 the deadlock and aborts one of the transactions.

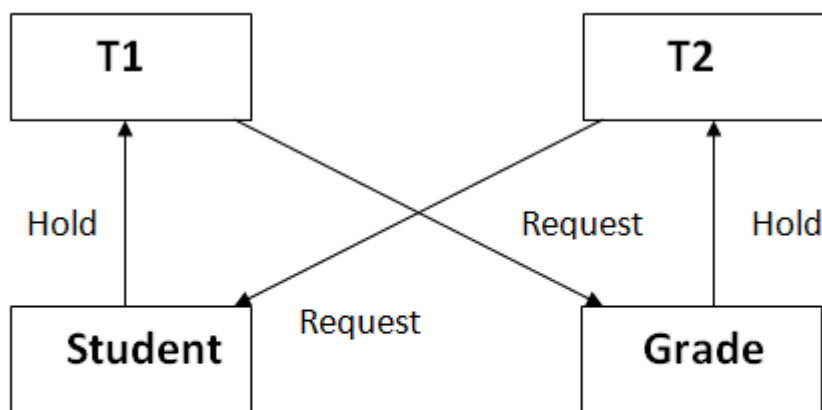


Figure: Deadlock in DBMS

Deadlock Avoidance

- When a database is stuck in a deadlock state, then it is better to avoid the database rather than aborting or restating the database. This is a waste of time and resource.
- Deadlock avoidance mechanism is used to detect any deadlock situation in advance. 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, deadlock prevention method can be used.

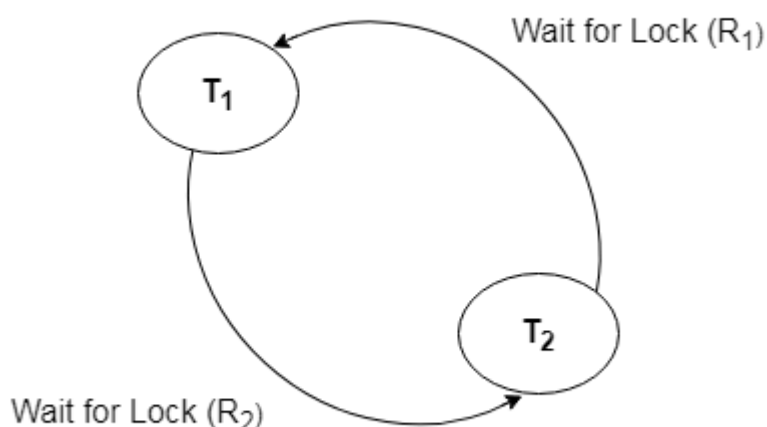
Deadlock Detection

In a database, when a transaction waits indefinitely to obtain a lock, then the DBMS should detect whether the transaction is involved in a deadlock or not. The lock manager maintains a Wait for the graph to detect the deadlock cycle in the database.

Wait for Graph

- This is the suitable method for deadlock detection. In this method, a graph is created based on the transaction and their lock. If the created graph has a cycle or closed loop, then there is a deadlock.
- The wait for the graph is maintained by the system for every transaction which is waiting for some data held by the others. The system keeps checking the graph if there is any cycle in the graph.

The wait for a graph for the above scenario is shown below:



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 allowed that transaction to be executed.

Wait-Die scheme

In this scheme, if a transaction requests for a resource which is already held with a conflicting lock by another transaction then the DBMS simply checks the timestamp of both transactions. **It allows the older transaction to wait until** the resource is available for execution.

Let's assume there are two transactions T_i and T_j and let $TS(T)$ is a timestamp of any transaction T . If T_2 holds a lock by some other transaction and T_1 is requesting for resources held by T_2 then the following actions are performed by DBMS:

1. Check if $TS(T_i) < TS(T_j)$ - If T_i is the older transaction and T_j has held some resource, then T_i is allowed to wait until the data-item is available for execution. That means if the older transaction is waiting for a resource which is locked by the younger transaction, then the **older transaction is allowed to wait for resource** until it is available.(wait)
2. Check if $TS(T_i) < TS(T_j)$ - If T_i is **older transaction and has held some resource** and if T_j is waiting for it, then **T_j is killed and restarted** later with the random delay

but with the same timestamp and T_i is allowed to continue as its older than T_j (die)

Wound wait scheme

- In wound wait scheme, if the older transaction requests for a resource which is held by the younger transaction, then **older transaction forces younger one to kill the transaction** and release the resource. After the minute delay, the younger transaction is restarted but with the same timestamp.(wound)
- If the older transaction has held a resource which is requested by the Younger transaction, then the younger transaction is asked to wait until older releases it.(wait)

Deadlock recovery:

- When deadlock is detected :
 - Some transaction will have to rolled back (made a **victim**) to break deadlock cycle.
 - Select that transaction as victim that will incur minimum cost
 - Rollback -- determine how far to roll back transaction
 - **Total rollback**: Abort the transaction and then restart it.
 - **Partial rollback**: Roll back victim transaction only as far as necessary to release locks that another transaction in cycle is waiting for
- Starvation can happen (why?)

- One solution: oldest transaction in the deadlock set is never chosen as victim