

Concurrency Control Mechanisms

The Concurrency Control Mechanism used for transform a schedule into serializable.

→ It is a process of managing multiple users access and modification in data simultaneously in shared or multi-user database systems.

How it helps?

1. **Data Consistency:** Ensures that data remains accurate and reliable despite concurrent access.
2. **Isolation:** Maintains the isolation property of transactions, so the outcome of a transaction is not affected by other concurrently executing transactions.
3. **Serializability:** Ensures that the result of concurrent transactions is the same as if the transactions had been executed serially

→

→ In serializability we check if a schedule is serializable or not but in concurrency control we use certain techniques to make a schedule serializable.

→ Why needed?

Why its needed?

- **Lost Updates:** When two or more transactions update the same data simultaneously, one of the updates might be lost. For example, if two users modify the same record at the same time, the changes made by one user could overwrite the changes made by the other. (WW)
- **Dirty Reads:** When a transaction reads data that has been modified by another transaction but not yet committed. If the first transaction rolls back, the other transaction will have read invalid data. (WR)
- **Uncommitted Dependency (Dirty Writes):** When a transaction modifies data that has been read by another transaction, leading to inconsistencies if one of the transactions rolls back.
- **Inconsistent Retrievals (Non-repeatable Reads):** Happens when a transaction reads the same data multiple times and gets different values each time because another transaction is modifying the data in between the reads.
- **Phantom Reads:** Occurs when a transaction reads a set of rows that satisfy a condition, but another transaction inserts or deletes rows that affect the set before the first transaction completes. This results in the first transaction reading different sets of rows if it re-executes the query.

Some common concurrency control mechanisms are:

- Lock-Based Protocols (2PL, Strict 2PL, Rigorous 2PL, Conservative 2PL)
- Timestamp-Based Protocols (Basic Timestamp Ordering (TO), Thoma's Write Rule)
- Optimistic Concurrency Control (OCC)
- Multi-version Concurrency Control (MVCC)

Locked Based Protocols

Types of Locks

1. **Binary Locks:** A simple mechanism where a data item can be either locked (in use) or unlocked. If a thread tries to acquire the lock when it's already locked, it must wait until the lock is released by the thread currently holding it.

2. **Shared and Exclusive Locks:**

Shared Lock (S-lock): Allows multiple transactions to read a data item simultaneously but prevents any of them from modifying it. Multiple transactions can hold a shared lock on the same data item at the same time. (Only used for read). However, no transaction can acquire an exclusive lock on that item as long as one or more shared locks are held.

Exclusive Lock (X-lock): Allows a transaction to both read and modify a data item. When an exclusive lock is held by a transaction, no other transaction can read or modify the data item. No other transaction can acquire a lock on the same data item until the exclusive lock is released.

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

**Unlocking a lock: U(A)

While shared and exclusive locks are vital for maintaining **data integrity and consistency** in concurrent environments, they can introduce significant challenges in terms of performance, deadlocks, reduced concurrency, and system complexity.

◆ ◆

Concurrency control mechanisms

Drawbacks of shared-exclusive locks

Performance issues : Managing locks requires additional CPU and memory resources. The process of acquiring, releasing, and managing locks can introduce significant overhead

Concurrency issues : Exclusive locks prevent other transactions from accessing locked data, which can significantly reduce concurrency.

Starvation: Some transactions may be delayed if higher-priority transactions consistently acquire locks before them, leading to starvation where a transaction never gets to proceed.

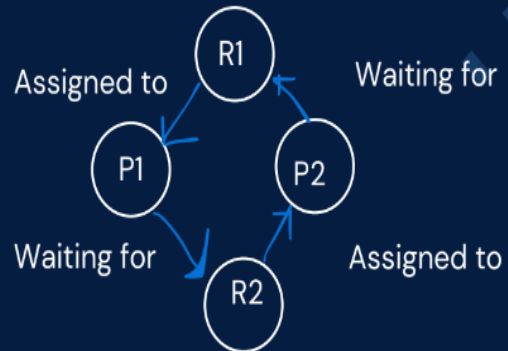
Deadlocks : Shared and exclusive locks can lead to deadlocks, where two or more transactions hold locks that the other transactions need.

Irrecoverable : If Transaction B commits after the lock is release based on a modified value in transaction A which fails after sometime.

Irrecoverable: Because Suppose Transaction A modifies and unlock a value which is now taken by Transaction B and B has now committed on basis of that value. But now if transaction A fails and rollbacks, B's taken value and committed in DB is inconsistent.

Deadlock : It is a situation when 2 or more transactions wait for one another to give up the locks.

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



2 Phase Locking (2PL): (advantage over locks)

Two-Phase Locking (2PL) : This protocol ensures serializability by dividing the execution of a transaction into two distinct phases

- **Growing Phase:** A transaction can acquire locks but cannot release any. This phase continues until the transaction has obtained all the locks it needs.
- **Shrinking Phase:** After the transaction releases its first lock, it can no longer acquire any new locks. During this phase, the transaction releases all the locks it holds.

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

Any transaction which is following 2PL locking achieves serializability and consistency.

In T1, the growing phase is till R(B), then U(A) and U(B) are in shrinking phase. T2 as A is locked by T1 which is in growing phase and cannot

release A, waits until T1 gets in its shrinking phase. **So, serializability is achieved. (ie. T1->T2).**

Two-Phase Locking (2PL)

Advantages :

- 1.It guarantees that the schedule of transactions will be serializable, meaning the results of executing transactions concurrently will be the same as if they were executed in some serial order.
- 2.By ensuring that transactions are serializable, 2PL helps maintain data integrity and consistency, which is critical in environments where data accuracy is essential.

Disadvantages :

- 1.Deadlocks, starvation and cascading rollbacks
- 2.Transactions must wait for locks to be released by other transactions. This can lead to increased waiting times and lower system throughput.
- 3.In case of a system failure, recovering from a crash can be complex

Cascading rollbacks happens when if T1 update any value A which is taken used by T2 and T3. After that when T1 fails and rollback instead of commit, T2 and T3 also need to rollback.

Strict 2 Phase Locking (strict 2PL): (advantage over 2PL)

Strict Two-Phase Locking (Strict 2PL):

A stricter variant where exclusive locks are held until the transaction commits or aborts. This helps prevent cascading rollbacks (where one transaction's rollback causes other transactions to roll back).

Advantages:

- Prevents Cascading Aborts
- Ensures Strict Serializability

Disadvantages:

- Since write locks are held until the end of the transaction, other transactions may be blocked for extended periods
- Transactions may experience longer wait times to acquire locks
- Deadlocks and starvation is there

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

Rigorous 2 Phase Locking:

LET'S START WITH DBMS :)

Concurrency control mechanisms

↓

Rigorous Two-Phase Locking:
An even stricter version where all locks (both shared and exclusive) are held until the transaction commits. This guarantees strict serializability.

Advantages:


- ✓ Since all locks are held until the end of the transaction, the system can easily ensure that transactions are serializable and can be recovered
- ✓ Prevents Cascading Aborts and Dirty Reads

Disadvantages:

- ✓ Performance bottlenecks
- ✓ Increased Transaction Duration
- ✓ Deadlocks and starvation is there

Handwritten notes on the right side of the slide:

- 2PL → X(A)
- 2PL → X(A) { → commit, → other
- R2PL → X(A)
- S(A)
- U(A)



*strict 2PL has only exclusive lock held until commit where else rigorous has both S and X held until commit.

Conservative 2 Phase Locking: (Static Two-Phase Locking)

- Conservative Two-Phase Locking(Static Two-Phase Locking) is a variant of the standard 2PL protocol that aims to **prevent deadlocks** entirely by requiring a transaction to acquire all the locks it needs before it begins execution.
- If the transaction is unable to acquire all the required locks (because some are already held by other transactions), it waits and retries. The transaction only starts execution once it has successfully acquired all the necessary locks.
- Since a transaction never starts executing until it has all the locks it needs, deadlocks cannot occur because no transaction will ever hold some locks and wait for others.

- In this scenario, deadlocks cannot occur because neither T1 nor T2 starts execution until it has all the locks it needs.

Here's how it works:

- When T1 begins, it checks if it can acquire **all the locks** it needs (both A and B in this case).
- If **both A and B are available**, T1 will acquire both locks.
- If **either A or B is not available**, T1 does **not acquire any locks at all** and waits until all the required locks (A and B) are available simultaneously.

Timestamp Based Protocols

- Ensure the serializability of transactions without using locks.
- Every transaction is assigned a **unique timestamp** when it starts or enters the system.
- **Older transactions have smaller timestamps, and younger transactions have larger timestamps.**
- This timestamp determines the transaction's position in the overall schedule.
- Older Transaction T1 should be scheduled first, then younger transaction T2 should be scheduled. (on a same data) **T1->T2**
- **W-Timestamp(X) and R-Timestamp(X):**

For each data item X, two timestamps are maintained:

- **W-Timestamp(X):** The **largest timestamp** of any transaction that successfully **wrote** to X.
- **R-Timestamp(X):** The **largest timestamp** of any transaction that successfully **read** from X.
- When a transaction tries to perform a read or write, the timestamp-based protocol (rules) checks if it is allowed based on the transaction's timestamp and the timestamps of the data item.
- Rules:

Timestamp Based Protocols: It assigns a unique timestamp

Rules : Consider if there are transactions performed on data item A.

$R_TS(A)$ \rightarrow Read time-stamp of data-item A.

$W_TS(A)$ \rightarrow Write time-stamp of data-item A.

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

- If $W_TS(A) > TS(T_i)$ then the operation is rejected. (rollback T_i)
- If $W_TS(A) \leq TS(T_i)$ then the operation is executed. (set $R_TS(A)$ as the

max of $(R_TS(A), TS(T_i))$

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

- If $TS(T_i) < R_TS(A)$ then the operation is rejected. (rollback T_i)
- If $TS(T_i) < W_TS(A)$ then the operation is rejected and T_i is rolled back otherwise the operation is executed. Set $W_TS(A) = TS(T_i)$

Here the rules are as such that older T_1 have all its read/write on data item A before younger T_2 , as to achieve serializability $T_1 > T_2$. This means T_1 should complete all its operations then only T_2 will do on that same data. T_2 should be dependent on T_1 as T_2 is executed after T_1 on that same data. (though they can run concurrently on different data).

The aborted transaction (in this case, T_1) is **restarted** with a **new timestamp**, meaning that its original timestamp is discarded, and it will now be treated as a newer transaction (with a larger timestamp than before).

