Module 6

12 March 2025 11:14

DBMS Concurrency Control

- Concurrency Control is the management procedure that is required for controlling concurrent execution of the operations that take place on a database.

But before knowing about concurrency control, we should know about concurrent execution.

Concurrent Execution in DBMS

- In a multi-user system, multiple users can access and use the same database at one time, which is known as the concurrent execution of the database. It means that the same database is executed simultaneously on a multi-user system by different users.

- While working on the database transactions, there occurs the requirement of using the database by multiple users for performing different operations, and in that case, concurrent execution of the database is performed.

- The thing is that the simultaneous execution that is performed should be done in an interleaved manner, and no operation should affect the other executing operations, thus maintaining the consistency of the database. Thus, on making the concurrent execution of the transaction operations, there occur several challenging problems that need to be solved.

Problems with Concurrent Execution

In a database transaction, the two main operations are **READ** and **WRITE** operations. So, there is a need to manage these two operations in the concurrent execution of the transactions as if these operations are not performed in an interleaved manner, and the data may become inconsistent. So, the following problems occur with the Concurrent Execution of the operations:

Problem 1: Lost Update Problems (W - W Conflict)

The problem occurs when two different database transactions perform the read/write operations on the same database items in an interleaved manner (i.e., concurrent execution) that makes the values of the items incorrect hence making the database inconsistent.

For example:

Consider the below diagram where two transactions TX and TY, are performed on the same account A where the balance of account A is $300.

Time	T_x	T_y
t_1	READ (A)	—
t_2	A = A - 50	
t_3	—	READ (A)
t_4	—	A = A + 100
t_5	—	—
t_6	WRITE (A)	—
t_7		WRITE (A)

LOST UPDATE PROBLEM

- At time t1, transaction TX reads the value of account A, i.e., $300 (only read).
- At time t2, transaction TX deducts $50 from account A that becomes $250 (only deducted and not

updated/write).

- Alternately, at time t3, transaction TY reads the value of account A that will be $300 only because TX didn't update the value yet.
- At time t4, transaction TY adds $100 to account A that becomes $400 (only added but not updated/write).
- At time t6, transaction TX writes the value of account A that will be updated as $250 only, as TY didn't update the value yet.
- Similarly, at time t7, transaction TY writes the values of account A, so it will write as done at time t4 that will be $400. It means the value written by TX is lost, i.e., $250 is lost.

  Hence data becomes incorrect, and database sets to inconsistent.

Dirty Read Problems (W-R Conflict)
The dirty read problem occurs when one transaction updates an item of the database, and somehow the transaction fails, and before the data gets rollback, the updated database item is accessed by another transaction. There comes the Read-Write Conflict between both transactions.

For example:

Consider two transactions TX and TY in the below diagram performing read/write operations on account A where the available balance in account A is $300:

Time	T_x	T_y
t_1	READ (A)	—
t_2	A = A + 50	—
t_3	WRITE (A)	—
t_4	—	READ (A)
t_5	SERVER DOWN ROLLBACK	—

DIRTY READ PROBLEM

- At time t1, transaction TX reads the value of account A, i.e., $300.
- At time t2, transaction TX adds $50 to account A that becomes $350.
- At time t3, transaction TX writes the updated value in account A, i.e., $350.
- Then at time t4, transaction TY reads account A that will be read as $350.
- Then at time t5, transaction TX rollbacks due to server problem, and the value changes back to $300 (as initially).
- But the value for account A remains $350 for transaction TY as committed, which is the dirty read and therefore known as the Dirty Read Problem.

Unrepeatable Read Problem (W-R Conflict)
Also known as Inconsistent Retrievals Problem that occurs when in a transaction, two different values are read for the same database item.

For example:

Consider two transactions, TX and TY, performing the read/write operations on account A, having an available balance = $300. The diagram is shown below:

Time	T_x	T_y
t_1	READ (A)	—
t_2	—	READ (A)
t_3	—	A = A + 100
t_4	—	WRITE (A)
t_5	READ (A)	—

UNREPEATABLE READ PROBLEM

- At time t1, transaction TX reads the value from account A, i.e., $300.
- At time t2, transaction TY reads the value from account A, i.e., $300.
- At time t3, transaction TY updates the value of account A by adding $100 to the available balance, and then it becomes $400.
- At time t4, transaction TY writes the updated value, i.e., $400.
- After that, at time t5, transaction TX reads the available value of account A, and that will be read as $400.
- It means that within the same transaction TX, it reads two different values of account A, i.e., $ 300 initially, and after updation made by transaction TY, it reads $400. It is an unrepeatable read and is therefore known as the Unrepeatable read problem.

Thus, in order to maintain consistency in the database and avoid such problems that take place in concurrent execution, management is needed, and that is where the concept of Concurrency Control comes into role.

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.

Challenges in Concurrent Transactions
Isolation is one of the major issues which the community has to consider to achieve data consistency, integrity and good performance of a DBMS when it supports concurrent transactions. These are a few of the major difficulties:

Data Consistency:

- **Lost Updates**: Happens when two or more transactions attempt to read the same record, modify it and then write it back, it would result in writing only one of the changes.

- **Temporary Inconsistencies**: Raise when a transaction is reading data that another transaction is simultaneously processing, resulting in interim or unpredictable values.

Data Integrity:

- **Non-Repeatable Reads**: Occur when one transaction retries a read operation and obtains a different value than before because another transaction has updated the value in between.

- **Phantom Reads:** Happen when a transaction resubmit a query that returns a set of rows which meet a condition and discover that the set of rows has been changed by another transaction that was inserting or deleting rows.

Isolation Levels and Performance

- **Low Isolation Levels**: Read Uncommitted and Read Committed for example, makes the query run faster but at the same time, bring about the possibility of dirty reads and non-repeatable reads.

- **High Isolation Levels:** As such, we have techniques such as Repeatable Read and Serializable options that ensure data consistency but harm concurrency and system performance.

Deadlocks:

- **Detection and Resolution:** After that, to identify deadlocks, it is necessary to monitor ongoing transactions, which are interested in resources and their allocation; the computation of these is computationally intensive, means that resolving deadlocks require the aborting of one or several transactions; this has negative impacts on the system throughput and users satisfaction.

- **Prevention and Avoidance**: Policies or measures to address or minimize deadlocks in operating systems can reduce concurrency and hence underutilize resources.

Resource Contention

- **Lock Contention**: If there are many transactions that require the use of locks, there is a potential to have small raw transaction rates, and long waits where there is a large transaction rate.

- **Hardware Resource Constraints**: This generate contention issues on a limited number of CPUs, memory and I/O operation when there are a high number of concurrent transactions.

Concurrency Control Mechanisms

The use of locks is very important in DBMS concurrency control because it is used in the control of multiple transactions without allowing unsynchronized changes to the database. Some of the concepts that play an important role in these protocols are shared locks and exclusive locks in addition to two phase locking (2PL) and strict two phase locking.

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.

What is a Lock?

It is associated with each data item that represents the state of the item with respect to the possible operations that can be performed on it. Its value is used in a locking scheme for manipulation of the associated data item and to control concurrent access.

- Locking a data item being used by a transaction can prevent other transactions running simultaneously from using these locked data items. This is one of the most commonly used techniques to ensure serialization, manipulation of the value of a lock is called locking.

Types of Locks

Various types of locks are used to control concurrency. Depending on the type of lock, the lock manager grants or denies access to other operations on the same data item. **B**inary locks are simple and have less practical use. Shared and exclusive locks also called read/write locks have more locking capabilities and have large practical usage.

Binary locks

A binary lock can have only two values:

Locked
Unlocked

Which are represented by 1 or 0 for simplicity. Each data item has a separate lock associated with it. If the data item is locked then it cannot be accessed by database operations that request the data item and if the data item is unlocked then it can be accessed when requested.

Following two operations are associated with binary locking of a data item A.

Lock (A)
Unlock (A)

A transaction requests access to a data item by first locking the data item using the Lock() operation. While doing so, if another operation of another concurrent transaction tries to access the same data item, it is forced to wait until the transaction that locked the data item has unlocked the same data item using the Unlock() operation. When a given transaction has locked a data item completes all its operations with that data item, it automatically unlocks the data item so that other transactions can use it.

The following rules must be followed whenever binary locking schemes are used:

Lock (): This operation must be issued by the transaction before any update operations such as a read or write performed on the transaction.

Unlock (): This operation must be issued by the transaction after all read or write operations in the transaction have completed.

A lock() operation cannot be released by a transaction if it already holds a lock on the data item.

An Unlock () operation cannot be issued by the transaction unless it already holds a lock on the data item.

Consider two transactions T1 and T2 both update the account balance by Rs 200 and Rs 300 respectively and the possible schedules is shown if these transactions are made to run concurrently.

T1	T2	Schedule
Read (A)	Read (A)	T1: Read (A)
A: = A + 200	A: = A + 300	T2: Read (A)
Write (A)	Write (A)	T1: A: = A + 200
		T1: Write (A)
		T2: A: = A + 300
		T2: Write (A)

Thus the lost update problem occurs in this schedule. Now if we apply binary lock on the transactions then this will happen.

T1	T2	Schedule
Lock (A)	Lock (A)	T1: Lock (A)
Read (A)	Read (A)	T1: Read (A)
A: = A + 200	A: = A + 300	T1: A: = A + 200
Write (A)	Write (A)	T1: Write (A)
Unlock (A)	Unlock (A)	T1: Unlock (A)
		T2: Lock (A)
		T2: Read (A)
		T2: A: = A + 300
		T2: Write (A)
		T2: Unlock (A)

Thus it is clear that a schedule consisting of transactions T1 and T2 will be serializable. This is because if T1 locks account A before T2 does then T2 cannot proceed until the transaction T1 unlocks the account A.

Similarly, if T2 locks account A before T1 does then T1 cannot proceed further until the transactions T2 unlocks the account A.

In binary locking method at most one of the transactions can hold a lock on a particular data item. Thus, no two transactions can access the same item.

There are two types of lock:

1. Shared lock:

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:

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

The following rules must be followed whenever shared/exclusive locks are used:

- A shared/exclusive lock must be applied on a data item in a transaction before a read operation is performed in the transaction.
- An exclusive lock must be applied on the data item in a transaction before any write operation on the data item in the transaction have completed.
- A transaction must issue an unlock operation on a data item after all read/write operations on the data item in the transaction have completed.
- A transaction must not execute an unlock operation unless it already holds a shared/exclusive lock on the data item.
- A transaction that already holds a shared/exclusive lock on the data item cannot issue a shared lock on the same data item. Similarly, a transaction that already holds a shared/exclusive lock on the data item cannot issue an exclusive lock on the same data item.

The compatibility relation between shared, exclusive and unlock is given be the Matrix as shown below. In

this matrix, we assume that the request for locking is made by the transaction that does not already hold a lock on the data item.

	Shared	Exclusive	Unlock
Shared	Yes	No	Yes
Exclusive	No	No	Yes
Unlock	Yes	Yes	--

The above matrix can be explained as follows:

If the current state of locking on a data item is shared such as a transaction has already applied a share lock on data item X then another transaction can also apply a shared lock on the same data item X i.e. lock mode of request can be shared. But it cannot apply exclusive lock on data item X which is already in shared lock by some other transaction. However you can unlock the data item X that is being applied shared locked by some transaction.
If the current state of locking on a data item is exclusive i.e. a transaction has already issued an exclusive lock on data item X then other transaction cannot issue a shared or exclusive lock on the same data item X. However, we can unlock the data item X which is being locked exclusively by the same transaction.
If the data item X is not locked i.e. it is unlocked then it can be locked using shared/ exclusive lock. However, an unlocked data item cannot be unlocked again.

Now consider the situations where two banking accounts A and B exist and let their balances be Rs 1000 and Rs 900. Let us take two transactions creating balance in accounts and another transaction transfers Rs. 200 from account A to account B.

T1	T2
Lock X(A)	Lock X(Sum)
Read (A)	Sum: = 0
A: = A - 200	Lock S(A)
Write (A)	Read (A)
Unlock (A)	Sum: = Sum + A
Lock X(B)	Unlock (A)
Read (B)	Lock S (B)
B: = B + 200	Read (B)
Write (B)	Sum: = Sum + B
nlock (B)	Write (Sum)
	Unlock (B)
	Unlock (Sum)

If the above two transactions T1 and T2 are executed in a serial order like T1 and then T2 or T2 and then T1 then the transaction T2 will display a value of Rs 1900 each time. Here Lock X means exclusive lock and Lock S means shared lock. If T1 and T2 transactions are performed simultaneously then schedule as shown below.

Schedule

T2: Lock X(Sum)
T2: Sum: = 0
T2: Lock S(A)
T2: Read (A)
T2: Sum: = Sum + A
T2: Unlock (A)
T1: Lock X(A)
T1: Read (A)
T1: A: = A - 200
T1: Write (A)
T1: Unlock (A)
T1: Lock X(B)
T1: Read (B)
T1: B: = B + 200
T1: Write (B)
T1: Unlock (B)
T2: Lock S (B)
T2: Read (B)
T2: Sum: = Sum + B
T2: Write (Sum)
T2: Unlock (B)
T2: Unlock (Sum)

In the above schedule, the value of the sum of the balances of accounts A and B comes out to be Rs 2100 which is incorrect. This locking scheme did not solve the inconsistent read problem because after these transactions were executing simultaneously, an inconsistent state of the database was created because the value of account A was added to the 'sum' before the modifications were made to the balance of account A.

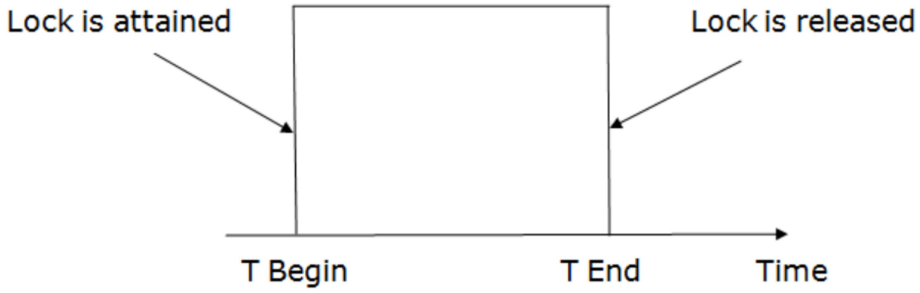
This schedule is a non-serializable schedule and a non-serializable schedule occurred even though both the transactions locked and unlocked the data items they used. Thus, locking can help in achieving serializibilty but it does not guarantee it.

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.

2. Pre-claiming Lock Protocol

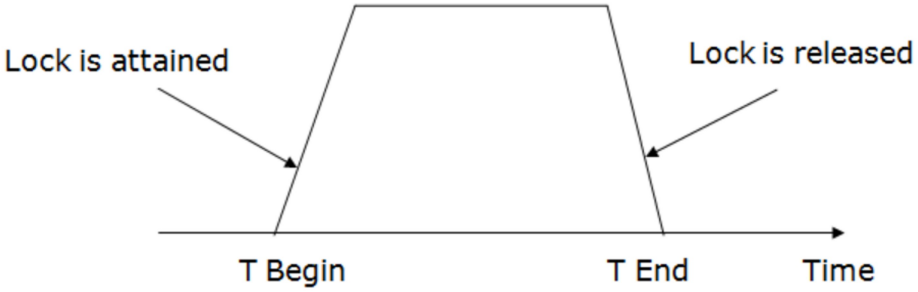
- Pre-claiming Lock Protocols evaluate the transaction to list all the data items on which they need locks.
- Before initiating an execution of the transaction, it requests DBMS for all the lock on all those data items.
- If all the locks are granted then this protocol allows the transaction to begin. 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 part, when the execution of the transaction starts, it seeks permission for the lock it requires.
- In the second part, the transaction acquires all the locks. The third phase is started as soon as the transaction releases its first lock.
- In the third phase, 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:

Upgrading of lock (from S(a) to X (a)) is allowed in growing phase.
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:

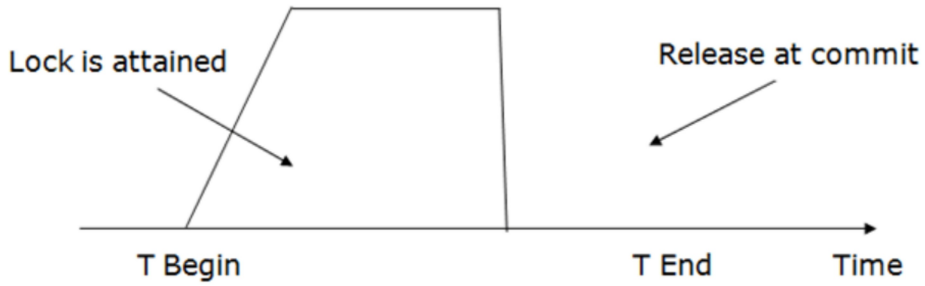
Growing phase: from step 1-3
Shrinking phase: from step 5-7
Lock point: at 3

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.

Thomas write Rule

Thomas Write Rule provides the guarantee of serializability order for the protocol. It improves the Basic Timestamp Ordering Algorithm.

The basic Thomas write rules are as follows:

If TS(T) < R_TS(X) then transaction T is aborted and rolled back, and operation is rejected.
If TS(T) < W_TS(X) then don't execute the W_item(X) operation of the transaction and continue processing.
If neither condition 1 nor condition 2 occurs, then allowed to execute the WRITE operation by transaction Ti and set W_TS(X) to TS(T).
If we use the Thomas write rule then some serializable schedule can be permitted that does not conflict serializable as illustrate by the schedule in a given figure:

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

Figure: A Serializable Schedule that is not Conflict Serializable

In the above figure, T1's read and precedes T1's write of the same data item. This schedule does not conflict serializable.

Thomas write rule checks that T2's write is never seen by any transaction. If we delete the write operation in transaction T2, then conflict serializable schedule can be obtained which is shown in below figure.

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

Figure: A Conflict Serializable Schedule