Unit 3

Concurrency Control

1. Introduction

A database system must always ensure proper execution of transactions, irrespective of any kinds of failure that may occur. It must also manage concurrent execution of transactions in such a way that it eliminates inconsistency of data within the database.

When several transaction are executed concurrently, it has been observed that isolation, which is one of the fundamental property of transactions is not preserved. This property can be preserved by controlling the interaction among the concurrent transaction. This control is done by using different mechanisms called concurrency control mechanisms or schemes. All these scheme ensure the serializability property of the schedules.

2. Ensuring Serializability by Locks

Theoretically, the concept of serializability ensures correct execution of concurrent transactions; thereby ensuring the consistency of the database. But practically, there are several methods that are implemented in a database system to ensure serializability. Two of these methods are

1. Methods / algorithms based on the concept of Locking.
2. Methods / algorithms based on the concept of timestamps.

These methods/algorithms stated above are called concurrency control algorithms.

All these above methods ensure the serializability property of the schedules.

In this section, we consider the management of concurrently executing transactions, by using different schemes.

1

Oct. 2017 – 3M
State different modes of locks that can be applied to a database item.

2.1 Lock Based Protocols

The most common way to ensure serializability is by making sure that data items are accessed in a mutually exclusive manner, i.e., when one transaction is accessing a data item, no other transaction can modify that data item. The most common method to ensure this is by use of locks. That is, a transaction is allowed to access a data item only if its currently holding a lock on that data item.

Types of Locks

Types of Locks
i. Binary lock
ii. Shared and Exclusive lock

Several types of locks can be used in concurrency control.

i. **Binary Lock:** A binary lock has two states or values. They are locked and unlocked (1 or 0). A distinct lock is associated with each database item X. If the value of lock of X is 1 ⇒ X is locked and if value of lock of X = 0, then the item can be accessed by any transaction.

The value of lock associated with a data item X is denoted by lock (X).

Two operations lock–item and unlock–item are included in a transaction, when using binary locking technique.

A transaction requests access to an item X using the lock–item(X) operation.

If lock (X) = 1, then the transaction is forced to wait, otherwise the lock is granted to the transaction for item X and it sets the value of lock (X) = 0.

When the transaction is finished using the item X, it issues an unlock–item (X) operator, which sets lock (X) = 0, so that it can be accessed by other transactions.

A binary lock enforces a mutual exclusion on the data item.

Lock–item and unlock–item operations are to be implemented as atomic operations that is, no interleaving should be allowed, once a lock or an unlock operation is started until the operation terminates or the transaction waits.

In case of binary locking, atmost one transaction can hold lock on a particular item, i.e., no two transactions can access the same item concurrently.

Thus even if 2 transactions want to read the value of a data item X, it cannot be done concurrently, even though multiple transactions reading the value of a data item concurrently do not lead to inconsistency in the database. This is a major drawback in Binary locking mechanism.

ii. **Shared and Exclusive lock:** In this technique, the above disadvantage of binary locking being too restrictive is removed by the lock variable for a data item X to have 3 states or values as-read–locked, write–locked, unlocked.

Thus, a transaction can request any of the following 3 operations: read–lock(X), write–lock(X), unlock(X). *A read–locked item* is also called sharelocked, since multiple transactions are allowed to read a database item concurrently. A *write–locked item* is called exclusive locked because a transaction exclusively holds the lock on an item, until it finishes updating the item. The following rules are to be enforced by using the multiple–mode locking technique.

a. A transaction T must issue the operation read–lock(X) or write–lock(X) before any read–item(X) operation is performed by T.

b. A transaction T must issue the operation write–lock(X) before any write–item(X) operation is performed by T.

c. A transaction T must issue the operation unlock(X) after all read–item(X) and write–item(X) operations are completed in T.

d. A transaction will not issue a read–lock(X) operation if it already holds a shared (read) lock on X or a write (exclusive) lock on X. Similar is the case of write–item(X).

e. A transaction will not issue an unlock(X) operation unless it holds a read or a write lock on X.

Upgradation of locks is allowed, i.e., it is possible for a transaction T to issue a read–lock(X) and then later on, to upgrade the lock by issuing a write–lock(X) operation.

If T is the only transaction holding a read lock on X at the time, it issues a write–lock operation, then the upgradation of lock is allowed.

Similarly, a downgrading of lock from write–lock to read–lock is possible for a transaction T. Using multiple mode locking does not guarantee serializability of schedules in which the transactions participate. *Fig. 3.1* shows an *example* where the above locking rules are followed, but a nonserializable schedule may still result.

This is because the database item Y in T_1 and X in T_2 were unlocked too early.

	T_1	T_2	
	slock(Y)		
	read–item(Y)		
Early unlocking of Y←	unlock(Y)		
		slock(X)	
		read–item(X)	
		unlock(X)	→ Early unlocking of X
		Xlock(Y)	
		read–item(Y)	
		Y :=X + Y	
		write–item(Y)	
		unlock(Y)	
	Xlock(X)		
	read–item(X)		
	X := X + Y		
	write–item(X)		
	unlock(X)		

Figure 3.1: A non–serializable schedule 'S' that uses locks

Thus by using multimode locking alone, serializability is not guaranteed. Hence, some additional protocol is required along with the multimode locking concerning the positioning of locking and unlocking operations in every transaction. This protocol is called *2–phase locking protocol*.

Note slock(X) means read–lock item(X); Xlock(X) ⇒ write lock(X).

iii. **Starvation of lock:** Suppose that transaction T_2 has a shared mode lock on data item and T_1 requests an exclusive mode lock on the same item. Clearly, T_1 has to wait for T_2 to release the shared mode lock.

Meanwhile, suppose that T_3–requests a shared mode lock on the same data item. This lock request is compatible with lock granted to T_2, so T_3 may be granted the shared mode lock.

At this point, T_2 may release the lock but still T_1 has to wait for T_3. But again, there may be a new transaction T_4 requesting a shared mode lock on the same data–item. It is possible that there is a sequence of transactions that each requests a shared mode lock on same data item and T_1 never gets the exclusive mode lock on the data item. The transaction T_1 may never make progress and is said to be starved.

Starvation of transactions can be avoided by granting locks as follows. When a transaction T_i requests a lock on a data item Q in a particular mode M, the lock is granted provided that,

a. There is no other transaction holding a lock on Q in a mode that conflict with M.

b. There is no other transaction that is waiting for a lock on Q, and that made its lock request before T_i.

2.2 Two Phase Locking Protocol (2PL)

A transaction is said to follow two phase locking protocol, if all locking operations (slock, xlock) precede the first unlock operation in the transaction.

In 2PL, the transactions can be divided into 2 phases: an *expanding* (or growing) *phase*, during which new locks can be acquired but none can be released and a *shrinking phase*, during which existing locks can be released, but no new locks can be acquired.

As per the above definition, upgradation of locks is allowed in the growing phase only. But if downgrading of locks is also allowed, then downgrading must be done in the shrinking phase. *For example:* a read–lock(X) operation that downgrades an already held write lock on X can appear only in the shrinking phase of the transaction.

In *fig. 3.1*, transactions T_1 and T_2 don't follow 2PL, since in T_1, it unlocks Y and then requests for exclusive lock on X.

If we enforce 2PL for the transactions T_1 and T_2 of *fig. 3.1*, then it can be rewritten as T_1 and T_1 as shown in *fig. 3.2.*

S: T_1	T_2
slock(Y)	
read–item(Y)	
xlock(X)	
unlock(Y)	
	slock(X)
	read–item(X)
	xlock(Y)
	unlock(X)
	read–item(Y)
	Y := Y + X
	write–item(Y)
	unlock(Y)
read–item(X)	
X = X + Y	
write–item (X)	
unlock(X)	

Figure 3.2: A schedule S with T_1, T_2 that follows 2PL

If every transaction in a schedule follows the 2PL protocol, the schedule is guaranteed to be serializable, removing the need to test for serializability of schedule any more. Thus, the lockup mechanism by enforcing 2PL also ensures serializability.

1
Apr. 2018 – 4M
Explain how 2PL ensures serializability?

The disadvantage of basic 2PL is that 2PL limits the amount of concurrency that occurs in a schedule. This is because a transaction can't release a lock on a data item X, even if it has finished using it, since it may be requiring new locks on other items later on. Hence, other transactions requiring lock on X have to wait until T unlocks it. Thus the data item X also remains locked with T, even though its not using it.

Similarly, a transaction T may lock an item Y earlier than it's needed-thus, another transaction seeking access to Y has to wait even though T is not using Y yet. This is the price for guaranteeing serializability of all schedules without having to check the schedules themselves.

2PL and Its Variations

There are a number of variations of 2PL.

i. **Conservative 2PL:** Also called a *static 2PL,* which requires a transaction to lock all the items it accesses before the transaction begins execution by predeclaring its read set and write set. Read set of a transaction is the set of all items that the transaction reads and write set of a transaction is the set of all items that the transaction writes.

1

Oct. 2018 – *4M*
Explain the two-phase locking protocol and list its variants.

If any of the predeclared items needed can't be locked, the transaction doesn't lock any item. Instead, it waits until all the items are available for locking. Conservative 2PL is a deadlock free protocol.

ii. **Strict 2PL:** This is the most popular variation of 2PL, which guarantees strict schedules. Strict 2PL requires that in addition to locking being 2 phased all exclusive mode locks taken by a transaction must be held until that transaction commits. This requirement ensures that any data written by uncommitted transactions is locked in exclusive mode until the transaction commits, thus, preventing any other transaction from reading the data. Strict 2PL also avoids cascading rollbacks.

iii. **Rigorous 2PL:** This variation of 2PL requires that all locks taken by a transaction should be held until the transaction commits. In case of rigorous 2PL, the transactions can be easily serialized in the order in which they commit. *Example* of basic 2PL is as follows:

Example

The following two transactions are two phase transactions.

T_1 :	lock X(B)	T_2 :	lock S(A)
	Read(B)		read(A)
	B = B – 50;		lock S(B)
	write(B) ;		read(B)
	lock X(A)		display(A + B)
	read(A)		unlock(A)
	A = A + 50		unlock(B)
	write(A)		
	unlock(B)		
	unlock(A)		

Two phase locking does not ensure freedom from deadlock. T_1 and T_2 are two phase transactions but they are **deadlocked.**

T_1	T_2
lock X(B)	
read(B)	
B = B – 50	
write(B)	
	lock S(A)
	read(A)
	lock S(B)
lock X(A)	
read(A)	
A = A + 50	
write(A)	
unlock(B)	
unlock(A)	
	read(B)
	display(A + B)
	unlock(A)
	unlock(B)

Here, T_1 has a X lock and T_2 wants a S on B and T_2 has a S lock and T_1 wants a X on A.

Cascading rollback may occur under 2-phase locking.

3. Basic Timestamp Method for Concurrency

Lock based protocols, described in the previous section determine the order between every pair of conflicting transaction at execution time, by the first lock that both members of pair request that involves incompatible modes.

1 **Oct. 2018 – *3M***
Explain the Basic timestamp ordering method for concurrency control.

In this section, we discuss another method for ensuring serializability of schedules, by providing an ordering of transactions in advance. The most common method is the Time Stamp ordering method.

Timestamp

Timestamp is an unique number that is given to each transaction in the systems denoted by $T_S(T_i)$. This is assigned by the database system, before the start of the transaction execution.

1 **Oct. 2017 – *1M***
What is timestamp?

If a transaction T_i is assigned time stamp as $T_s(T_i)$ and a new transaction T_j enters the system, then $T_S(T_i) < T_s(T_j)$.

Here, T_i becomes the older transaction and T_j becomes the younger transaction.

There are two simple methods for implementing timestamps:

i. The value of the system clock can be used as the timestamp. So, at whatever time the transaction enters the system, that time becomes its timestamp.

ii. By using a logical counter, whose value is incremented after a new timestamp has been assigned.

The serializability order is determined by the timestamp of the transactions. Thus if $T_s(T_i) < T_s(T_j)$, then the system must ensure that the produced schedule is equivalent to a serial schedule in which the transaction T_i appears before transaction T_j.

In addition, each data item Q of the database is assigned two timestamp values:

i. Wtimestamp(Q) denoted by WS(Q), denotes the largest timestamp of any transaction that executes write(Q) successfully.

ii. Rtimestamp(Q) denoted by RS(Q), denotes the largest timestamp of any transaction that executes read(Q) successfully.

These timestamps of data items are updated whenever a new read or write operation is executed on Q.

3.1 The Basic Time-Stamp Ordering Protocol

The basic Timestamp ordering protocol guarantees that all conflicting read and write operations are executed in timestamp order. The protocol is given as follows:

i. Consider that transaction T_i issues a read(Q) operation.

Then this operation is allowed only if the $T_S(T_i) \geq W_S(Q)$ and $R_S(Q)$ is set to maximum of $R_S(Q)$ and $R_S(T_i)$. Because, if $T_S(T_i) < W_S(Q)$ then it implies that T_i is an old transaction that is trying to read the value of Q, written by the most recent transaction. Thus, it implies that T_i needs to read a value of Q that was already overwritten. In such cases T_i is rolled back and read operation of T_i is rejected.

ii. Consider that transaction T_i issues a write operation on data item Q; then, this write operation is valid and allowed only if the following two conditions are true:

a. $T_S(T_i) > R_S(Q)$, since if $T_S(T_i) < R_S(Q)$ then it implies that some recent or younger transaction has already read the value of Q and hence, T_i is trying to write a Q that was previously needed by the system, and now the system has assumed that the value would never be produced.

b. $T_S(T_i) > W_S(Q)$ since if $R_S(T_i) < W_S(Q)$, then T_i is attempting to write an obsolete value of Q.

If above two conditions are met, then T_i is allowed to write the value of Q and $W_S(Q)$ is set to $T_S(T_i)$. Otherwise, T_i is rolled back and restarted again with a new timestamp.

To illustrate this protocol, an *example* of two transactions T_1 and T_2 is given below:

```
T1  :  read(x)
       read(y)
       display(x + y)
T2  :  read(y)
       x : = x – 50
       write(x)
       read(y)
       A = A + 50
       write(A)
       display(A + B)
```

The following is the schedule S_1 of the above two transactions T_1 and T_2, which is made possible under timestamp protocol. Here, $T_S(T_1) < T_S(T_2)$.

T_1	T_2
read(x)	
	read(x)
	x := x –50
	write(x)
read(y)	
	read(y)
display(x + y)	
	y := y + 50
	write(y)
	display(x + y)

Figure 3.3: Schedule S_1

In the above schedule, the transactions are given timestamps immediately before its first instruction.

The basic timestamp ordering protocol ensures conflict serializability, since the conflicting operations are processed in time stamp order.

Another advantage of the protocol is that the protocol is a no deadlock protocol, i.e., ensures freedom from deadlocks.

Deadlocks are removed, since no transaction waits for a data item. If a transaction is not able to get a data item, then it's rolled back and hence doesn't wait.

But disadvantage is that there is a possibility of starvation of long transactions, if a sequence of conflicting short transactions causes repeated restarting of the long transaction.

Another disadvantage is that the protocol may generate non-recoverable schedules.

2

Oct. 2018 – *1M*
State the Thomas write rule.
Apr. 2018 – *3M*
State and explain Thomas write rule.

3.2 Thomas Write Rule

A modification of timestamp protocol is the timestamp protocol using Thomas Write Rule. Thomas Write Rule provides greater concurrency than the basic timestamp protocol.

The Thomas Write Rule in its simplest form states that if a transaction issues a write(Q) and as per the basic timestamp ordering, if this transaction satisfies the first condition of the protocol, i.e., $T_s(T_i) \geq R_s(Q)$ and if it fails in the 2nd condition, i.e., $T_s(T_i) < W_s(Q)$, then this second condition can be overruled and T_i can be allowed to write(Q), instead of being aborted.

Let us consider an *example* to justify the Thomas Write Rule.

Consider the schedule S_2 of transactions T_1 and T_2 as follows:

T_1	T_2
read(x)	
	write(x)
	commit
write(x)	
commit	

Figure 3.4: Schedule S_2

Let's apply the basic timestamp protocol to the schedule. Here, $T_s(T_1) < T_s(T_2)$.

The instruction read(x) of T_1 is executed and $R_s(x)$ is set to $T_s(T_1)$.