DBMS Lecture 42 and 43
Need for Concurrency Control

Need for Concurrency Control

1. Lost Update Problem
2. Dirty Read Problem
3. Inconsistent Analysis Problem (Phantom Read Problem)

Lost Update Problem

Time	T1	T2	X
t1		Begin_transaction	100
t2	Begin_Transaction	Read(X)	100
t3	Read(X)	X : = X - 50;	100
t4	X : = X – 10;	Write (X)	50
t5	Write(X)	Commit;	90
t6	Commit;		90

Dirty Read Problem

Time	T1	T2	X
t1		Begin_transaction	100
t2	Begin_Transaction	Read(X)	100
t3		X : = X - 50;	100
t4		Write (X)	50
t5	Read(X)		50
t6	X : = X – 10;	Rollback;	100
t7	Write(X)		40
t8	Commit;		40

Inconsistent Analysis Problem

Time	T1	T2	X	Y	Z	Sum
t1	Begin_Transaction	Begin_transaction	100	50	25	
t2	Read(X)	Sum : = 0;	100	50	25	0
t3	X : = X – 10;	Read(X)	100	50	25	0
t4	Write(X)	Sum := Sum + X;	90	50	25	100
t5	Read(Z)	Read(Y)	90	50	25	100
t6	Z := Z + 10;	Sum := Sum + Y;	90	50	25	150
t7	Write(Z)		90	50	35	150
t8	Commit;	Read(Z)	90	50	35	150
t9		Sum := Sum + Z;	90	50	35	185
t10		Display(Sum)	90	50	35	185
t11		Commit;	90	50	35	185

Locking Protocols

1. Shared Lock [Lock-S(Q)] – If any transaction wants to read the value of Q then it must acquire the shared lock to perform read(Q).
2. Exclusive Lock [Lock-X (Q)] – If any transaction wants to perform write or read and write both on data item Q then it must acquire the exclusive lock to perform both read(Q) or write(Q).
3. Unlock (Q) – If transaction no longer need to perform both read(Q) or write (Q) operation then it can perform unlock (Q).

Lock Compatibility Matrix

Lock Compatibility Matrix	Shared	Exclusive
Shared	Yes	No
Exclusive	No	No

Solving Lost Update Problem using locks

Time	T1	T2	X
t1		Begin_Transaction	100
t2	Begin_Transaction	Lock – X (X)	100
t3	Lock – X (X)	Read(X)	100
t4	WAIT	X : = X – 50;	100
t5	WAIT	Write (X)	50
t6	WAIT	Unlock(X)	50
t7	Read(X)	Commit;	50
t8	X : = X – 10;		50
t9	Write(X)		40
t10	Unlock(X)		40
t11	Commit;		40

Solving Dirty Read Problem Using Locks

Time	T1	T2	X
t1		Begin_Transaction	100
t2	Begin_Transaction	Lock – X (X)	100
t3	Lock – X (X)	Read(X)	100
t4	WAIT	X : = X - 50;	100
t5	WAIT	Write (X)	50
t6	WAIT	Unlock(X)	50
t7	Read(X)		50
t8	X : = X – 10;	Rollback;	100
t9	Write(X)		40
t10	Unlock(X)		40
t11	Commit;		40

Two Phase Locking Protocol (2PL)

Locks can be acquired in 2 phases.

1. Growing Phase – Transactions can acquire locks but it cannot release locks.
2. Shrinking Phase – Transactions can release the locks but it cannot acquire new locks.

Two Phase Locking Protocol (2PL)

Three Types

1. Strict 2PL – It can release shared lock anytime but it cannot release exclusive lock until the transaction terminates (i.e. either committed or rolled back).
2. Rigorous 2PL – It cannot release any locks (Shared lock and exclusive lock) until the transaction terminates (i.e. either committed or rolled back).
3. Conservative 2PL – It can acquired locks in advance to all the data items that it wants to perform read or write and release the locks when the transaction terminates.

Solving Lost Update Problem using 2PL

Time	T1	T2	X
t1		Begin_transaction	100
t2	Begin_Transaction	Lock – X (X)	100
t3	Lock – X (X)	Read(X)	100
t4	WAIT	X : = X - 50;	100
t5	WAIT	Write (X)	50
t6	WAIT	Commit; / Unlock(X)	50
t7	Read(X)		50
t8	X : = X – 10;		50
t9	Write(X)		40
t10	Commit; / Unlock(X)		40

Solving Dirty Read Problem Using 2PL

Time	T1	T2	X
t1		Begin_Transaction	100
t2	Begin_Transaction	Lock – X (X)	100
t3	Lock – X (X)	Read(X)	100
t4	WAIT	X : = X - 50;	100
t5	WAIT	Write (X)	50
t6	WAIT		50
t7	WAIT		50
t8	WAIT	Rollback; / Unlock(X)	100
t9	Read(X)		100
t10	X : = X – 10;		100
t11	Write(X)		90
t12	Commit; / Unlock(X)		90

DeadLock

Time	T1	T2
t1	Begin_Transaction	Begin_transaction
t2	Read(X)	Read(Y)
t3	X : = X – 10;	Y := Y – 20;
t4	Write(X)	Write(Y)
t5	Read(Y)	Read(X)
t6	Y := Y + 10;	X := X + 20;
t7	Write(Y)	Write(X)
t8	Commit;	Commit;

Time	T1	T2
t1	Begin_Transaction	Begin_transaction
t2	Lock-X(X)	Lock-X(Y)
t3	Read(X)	Read(Y)
t4	X := X – 10;	Y := Y – 20;
t5	Write(X)	Write(Y)
t6	Lock-X(Y)	Lock-X(X)
t7	WAIT	WAIT
t8	WAIT	WAIT
t9	…	…
t10	…	…

2PL

- 2PL protocol ensures conflict serializablity according to the lock points but it cannot be freedom from deadlocks.
- Only conservative 2PL is free from deadlocks.

Timestamp

- With each transaction *Ti* in the system, we associate a unique fixed timestamp, denoted by TS(*Ti*).
- This timestamp is assigned by the database system before the transaction *Ti* starts execution.
- If a transaction *Ti* has been assigned timestamp TS(*Ti*), and a new transaction *Tj* enters the system, then TS(*Ti*) < TS(*Tj*).
- There are two simple methods for implementing this scheme:
  1. Use the value of the *system clock* as the timestamp; that is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system.
  2. Use a **logical counter** that is incremented after a new timestamp has been assigned; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system.

Timestamp

- The timestamps of the transactions determine the serializability order. Thus, if TS(*Ti*) < TS(*Tj*), then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction *Ti* appears before transaction *Tj* .
- To implement this scheme, we associate with each data item *Q* two timestamp values :
  1. **W-timestamp**(*Q*) → denotes the largest timestamp of any transaction that executed write(*Q*) successfully.
  2. **R-timestamp**(*Q*) → denotes the largest timestamp of any transaction that executed read(*Q*) successfully.
- These timestamps are updated whenever a new read(*Q*) or write(*Q*) instruction is executed.

Timestamp

- The timestamps of the transactions determine the serializability order. Thus, if TS(*Ti*) < TS(*Tj*), then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction *Ti* appears before transaction *Tj* .
- To implement this scheme, we associate with each data item *Q* two timestamp values :
  1. **W-timestamp**(*Q*) → denotes the largest timestamp of any transaction that executed write(*Q*) successfully.
  2. **R-timestamp**(*Q*) → denotes the largest timestamp of any transaction that executed read(*Q*) successfully.
- These timestamps are updated whenever a new read(*Q*) or write(*Q*) instruction is executed.

The Timestamp – Ordering Protocol

- The **timestamp-ordering protocol** ensures that any conflicting read and write operations are executed in timestamp order. This protocol operates as follows:

1. Suppose that transaction T_i issues read(Q).
   a) If TS(T_i) < W-timestamp(Q), then T_i needs to read a value of Q that was already overwritten. Hence, the read operation is rejected, and T_i is rolled back.
   b) If TS(T_i) ≥ W-timestamp(Q), then the read operation is executed, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(T_i).

The Timestamp – Ordering Protocol

2. Suppose that transaction *Ti* issues write(*Q*).

a) If TS(*Ti*) < R-timestamp(*Q*), then the value of *Q* that *Ti* is producing was needed previously, and the system assumed that that value would never be produced. Hence, the system rejects the write operation and rolls *Ti* back.

b) If TS(*Ti*) < W-timestamp(*Q*), then *Ti* is attempting to write an obsolete value of *Q*. Hence, the system rejects this write operation and rolls *Ti* back.

c) Otherwise, the system executes the write operation and sets W-timestamp(*Q*) to TS(*Ti*).

- A transaction *Ti,* that is rolled back by the concurrency-control scheme as result of either a read or write operation being issued is assigned a new timestamp and is restarted.

The Timestamp Ordering Protocol

T1	T2
read(B)	
	read(B)
	B = B – 50
	write(B)
read(A)	
	read(A)
display(A+B)	
	A = A + 50
	write(A)
	display(A+B)

RTS(A) = 0

WTS(A) = 0

RTS(B) = 0

WTS(B) = 0

The Timestamp Ordering Protocol

T1 = 1	T2 = 2 (6)	T3 = 3 (7)	T4 = 4	T5 = 5
				Read(X)
	Read(Y)			
Read(Y)				
		Write(Y)		
		Write(Z)		
				Read(Z)
	Read(Y or Z)			
Read(X)				
		Write(Z)		
				Write(Y)
				Write(Z)

RTS(X) = 0

WTS(X) = 0

RTS(Y) = 0

WTS(Y) = 0

RTS(Z) = 0

WTS(Z) = 0

Advantages of Timestamp – Ordering Protocol

1. The timestamp-ordering protocol ensures conflict serializability. This is because conflicting operations are processed in timestamp order.
2. The protocol ensures freedom from deadlock, since no transaction ever waits.

Disadvantages of Timestamp – Ordering Protocol

1. There is a possibility of starvation of long transactions if a sequence of conflicting short transactions causes repeated restarting of the long transaction.
2. The protocol can generate schedules that are not recoverable.

Thomas' Write Rule

- Thomas Write Rule provides a modification to the timestamp-ordering protocol that allows greater potential concurrency than does the timestamp – ordering protocol.
- Let us consider schedule and apply the timestamp-ordering protocol. Since T1 starts before T2, we shall assume that TS(T1) < TS(T2). The read(*Q*) operation of T1 succeeds, as does the write(*Q*) operation of T2.When T1 attempts its write(*Q*) operation, we find that TS(T1) < W-timestamp(*Q*), since W-timestamp(*Q*) = TS(T2). Thus, the write(*Q*) by T1 is rejected and transaction T1 must be rolled back.

T1	T2
read(Q)	
	write(Q)
write(Q)	

Thomas' Write Rule

- Although the rollback of T1 is required by the timestamp-ordering protocol, it is unnecessary. Since T2 has already written *Q*, the value that T1 is attempting to write is one that will never need to be read.
- Any transaction Ti with TS(Ti) < TS(T2) that attempts a read(*Q*) will be rolled back, since TS(Ti) < W-timestamp(*Q*). Any transaction Tj with TS(Tj) > TS(T2) must read the value of *Q* written by T2, rather than the value written by T2.
- This observation leads to a modified version of the timestamp-ordering protocol in which obsolete write operations can be ignored under certain circumstances.
- The protocol rules for read operations remain unchanged. The protocol rules for write operations, however, are slightly different from the timestamp-ordering protocol.

Thomas' Write Rule

- The modification to the timestamp-ordering protocol, called **Thomas' write rule**, is this:
- Suppose that transaction *Ti* issues write(*Q*).

a) If TS(*Ti*) < R-timestamp(*Q*), then the value of *Q* that *Ti* is producing was previously needed, and it had been assumed that the value would never be produced. Hence, the system rejects the write operation and rolls *Ti* back.

b) If TS(*Ti*) < W-timestamp(*Q*), then *Ti* is attempting to write an obsolete value of *Q*. Hence, this write operation can be ignored.

c) Otherwise, the system executes the write operation and sets W-timestamp(*Q*) to TS(*Ti*).

Thomas' Write Rule

- Thomas' write rule makes use of view serializability by, in effect, deleting obsolete write operations from the transactions that issue them.
- This modification of transactions makes it possible to generate serializable schedules that would not be possible under the other protocols.
- For example, schedule given is not conflict serializable and, thus, is not possible under any of two-phase locking, the tree protocol, or the timestamp-ordering protocol. Under Thomas' write rule, the write(*Q*) operation of T1 would be ignored. The result is a schedule that is view equivalent to the serial schedule <T1, T2>.

T1	T2
read(Q)	
	write(Q)
write(Q)	

GATE Questions

Which of the following concurrency control protocols ensure both conflict serializability and freedom from deadlock?

I. 2-phase locking

II. Time-stamp ordering

(A) I only

(B) II only

(C) Both I and II

(D) neither I nor II

[GATE 2010]

GATE Questions

For the schedule given below, which of the following is correct :

1	Read A	
2		Read B
3	Write A	
4		Read A
5		Write A
6		Write B
7	Read B	
8	Write B	

(A) This schedule is serialized and can occur in a scheme using 2PL protocol

(B) This schedule is serializable but cannot occur in a scheme using 2PL protocol

(C) This schedule is not serializable but can occur in a scheme using 2PL protocol

(D) This schedule is not serializable and cannot occur in a scheme using 2PL protocol.

[GATE 1999]

GATE Questions

Consider the following two phase locking protocol. Suppose a transaction T accesses (for read or write operations), a certain set of objects $\{O_1, \ldots, O_k\}$. This is done in the following manner :

Step 1. T acquires exclusive locks to $O_1, \ldots, O_k$ in increasing order of their addresses.

Step 2. The required operations are performed.

Step 3. All locks are released.

This protocol will

(A) guarantee serializability and deadlock-freedom

(B) guarantee neither serializability nor deadlock-freedom

(C) guarantee serializability but not deadlock-freedom

(D) guarantee deadlock-freedom but not serializability **[GATE 2016]**

GATE Questions

In a database system, unique time stamps are assigned to each transaction using Lamport's logical clock . Let TS(T1) and TS(T2) be the timestamps of transactions T1 and T2 respectively. Besides, T1 holds a lock on the resource R, and T2 has requested a conflicting lock on the same resource R. The following algorithm is used to prevent deadlocks in the database system assuming that a killed transaction is restarted with the same timestamp.

if TS(T2) < TS(T1) then

T1 is killed

else T2 waits.

Assume any transaction that is not killed terminates eventually. Which of the following is TRUE about the database system that uses the above algorithm to prevent deadlocks?

(A) The database system is both deadlock-free and starvation- free.

(B) The database system is deadlock- free, but not starvation-free.

(C) The database system is starvation-free but not deadlock- free.

(D) The database system is neither deadlock- free nor starvation-free.

[GATE 2017]

For Video lecture on this topic please subscribe to my youtube channel.

The link for my youtube channel is

https://www.youtube.com/channel/UCRWGtE76JlTp1iim6aOTRuw?sub_confirmation=1