Database Management System 22
Locking Protocols

Chittaranjan Pradhan
School of Computer Engineering,
KIIT University

Two-Phase Locking Protocol

Two-Phase Locking Protocol

Two-phase locking protocol is a protocol which ensures serializability

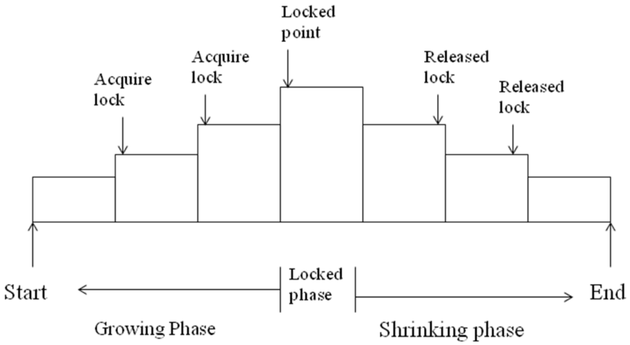
This protocol requires that each transaction issues lock and unlock requests in two phases. The two phases are:

- **Growing phase**: Here, a transaction acquires all required locks without unlocking any data, i.e. the transaction may not release any lock
- **Shrinking phase**: Here, a transaction releases all locks and cannot obtain any new lock

The point in the schedule where the transaction has obtained its final lock is called the lock point of the transaction

Transactions can be ordered according to their lock points

Two-Phase Locking Protocol...



- Two transactions cannot have conflicting locks
- No unlock operation can precede a lock operation in the same transaction
- No data are affected until all locks are obtained, i.e. until the transaction is in its locked point

Two-Phase Locking Protocol...

Two-Phase Locking Protocol

Two-phase locking does not ensure freedom from deadlock. Along with the serializability property, the schedules should be cascadeless in nature

Cascading rollback is possible under two-phase locking protocol

Schedule7

T_4	T_5	T_6
Lock-X(A);		
Read(A);		
Lock-S(B);		
Read(B);		
Write(A);		
Unlock(A);		
	Lock-X(A);	
	Read(A);	
	Write(A);	
	Unlock(A);	
		Lock-S(A);
		Read(A);

Two-Phase Locking Protocol...

- Cascading rollbacks can be avoided by a modification of the original two-phase locking protocol called strict two-phase locking protocol. In this protocol, a transaction must hold all its exclusive locks till it commits or aborts
- Rigorous two-phase locking protocol is even stricter, which requires that all the locks be held until the transaction commits or aborts. In this protocol, transactions can be serialized in the order in which they commit
- *Most database systems implement either strict or rigorous two-phase locking*

Lock Conversions

T_7
Read(a1);
Read(a2);
Read(a3);
...
Read(an);
Write(a1);

T_8
Read(a1);
Read(a3);
Display(a1+a3);

Lock Conversions

Lock Upgrade: This is the process in which a shared lock is upgraded to an exclusive lock

Lock Downgrade: This is the process in which an exclusive lock is downgraded to a shared lock

Lock upgrading can take place only in the growing phase, where as lock downgrading can take place only in the shrinking phase

Lock Conversions...

Thus, the two-phase locking protocol with lock conversions:

- ***First Phase***:
  - Can acquire a lock-S on item
  - Can acquire a lock-X on item
  - Can convert a lock-S to a lock-X (upgrade)
- ***Second Phase***:
  - Can release a lock-S
  - Can release a lock-X
  - Can convert a lock-X to a lock-S (downgrade)

Like the basic two-phase locking protocol, two-phase locking with lock conversion generates only conflict-serializable schedules and transactions can be serialized by their lock points

If the exclusive locks are held until the end of the transaction, then the schedules became cascadeless

Lock Conversions...

Schedule8

T_7	T_8
Lock-S(a1);	
Read(a1);	
	Lock-S(a1);
	Read(a1);
Lock-S(a2);	
Read(a2);	
Lock-S(a3);	
Read(a3);	
	Lock-S(a3);
	Read(a3);
	Display(a1+a3);
	Unlock(a1);
	Unlock(a3);
...	
Lock-S(an);	
Read(an);	
Upgrade(a1);	
Write(a1);	

Timestamp-Based Protocols

Timestamp-Based Protocols

The timestamp method for concurrency control doesn't need any locks and therefore this method is free from deadlock situation

Locking methods generally prevent conflicts by making transaction to wait; whereas timestamp methods do not make the transactions to wait. Rather, transactions involved in a conflicting situation are simply rolled back and restarted

A timestamp is a unique identifier created by the Database system that indicates the relative starting time of a transaction. Timestamps are generated either using the system clocks or by incrementing a logical counter every time a new transaction starts

Timestamp protocol is a concurrency control protocol in which the fundamental goal is to order the transactions globally in such a way that older transactions get priority in the event of a conflict

Timestamps

Timestamps

Timestamp TS(T_i) is assigned by the database system before the transaction T_i starts its execution

The timestamps of the transactions determine the serializability order. Thus, if TS(T_i) < TS(T_j), then the system must ensure that the produced schedule is equivalent to a serial schedule in which T_i appears before T_j

There are two timestamp values associated with each data item Q:

- **W-Timestamp(Q)**: It denotes the largest timestamp of any transaction that executed write(Q) operation successfully
- **R-Timestamp(Q)**: It denotes the largest timestamp of any transaction that executed read(Q) operation successfully

Timestamp Ordering Protocols

Timestamp Ordering Protocols

This ensures that any conflicting read and write operations are executed in timestamp order

- Suppose transaction T_i issues read(Q):
  - If TS(T_i) < W-TS(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
  - If TS(T_i) $\geq$ W-TS(Q), then the read operation is executed, and R-TS(Q) is set to maximum of R-TS(Q) and TS(T_i)
- Suppose transaction T_i issues write(Q):
  - If TS(T_i) < R-TS(Q), then the value of Q that T_i is producing was needed previously, and the system assumed that the value would never be produced. Hence, the system rejects the write operation and rolls T_i back
  - If TS(T_i) < W-TS(Q), then T_i is attempting to write an obsolete value of Q. Hence, the system rejects this write operation and rolls T_i back
  - Otherwise, the system executes the write operation and sets W-TS(Q) to TS(T_i)

Timestamp Ordering Protocols...

If a transaction T_i is rolled back by the concurrency-control scheme, the system assigns it a new timestamp and restarts it

T_1	T_2	T_1	T_2	T_1	T_2
Read(A);		Read(A);		Read(A);	
A:=A-100;		A:=A-100;		A:=A-100;	
Write(A);			Read(A);	Write(A);	
	Read(A);		Temp:=0.2*A;		Read(A);
	Temp:=0.2*A;		A:=A-Temp;		Temp:=0.2*A;
	A:=A-Temp;		Write(A);		A:=A-Temp;
	Write(A);		Read(B);	Read(B);	
Read(B);		Write(A);		B:=B+100;	
B:=B+100;		Read(B);			Write(A);
Write(B);		B:=B+100;			Read(B);
	Read(B);	Write(B);			B:=B+ Temp;
	B:=B+ Temp;		B:=B+ Temp;		Write(B);
	Write(B);		Write(B);	Write(B);	
Schedule3		Schedule4		Schedule5	

Timestamp Ordering Protocols...

From the above examples, we can say that the timestamp ordering protocol always ensures conflict serializability. This is because conflicting operations are processed in timestamp order

The protocol ensures freedom from deadlock, since no transaction ever waits

However, there is a possibility of starvation of long transactions if a sequence of conflicting short transactions causes repeated restarting of the long transaction

If a transaction is found to be getting restarted repeatedly; conflicting transactions need to be temporarily blocked to enable the transaction to finish

Problems with Timestamp Ordering Protocols

Problems with Timestamp Ordering Protocols

Suppose T_i aborts, but T_j has read a data item written by T_i. Then, T_j must abort; if T_j had been allowed to omit earlier, the schedule is not recoverable. Further, any transaction that has read a data item written by T_j must abort. This can lead to cascading rollback; i.e. chain of rollbacks

The solutions to this type of problem are:

- A transaction is structured in such a way that its writes are performed at the end of its processing. At the same time, all writes of a transaction form an atomic action; i.e. no transaction may execute while a transaction is being written, or
- Wait for the data to be committed before reading it

Thomas' Write Rule

Thomas' Write Rule

One of the problems with basic timestamp ordering protocol is that it does not guarantee recoverable schedules

A modification to the basic timestamp ordering protocol that relaxes the conflict serializability can be used to provide greater concurrency by rejecting obsolete write operations

Schedule9

T_9	T_{10}
Read(Q);	
	Write(Q);
Write(Q);	
Read(M);	
Write(M);	

Thomas' Write Rule...

Thomas' Write Rule...

- T_i issues Read(Q):
  - Remains same
- T_i issues write(Q):
  - If TS(T_i) < R-TS(Q), T_i is aborted, rolled back and is restarted with a new timestamp
  - If TS(T_i) < W-TS(Q), T_i is attempting to write an obsolete value of Q. Hence, this write operation can be ignored
  - Otherwise, the system executes the write operation and sets W-TS(Q) = TS(T_i)

Thomas' write rule makes use of view Serializability by deleting obsolete write operations from the transactions that issue them