Locking protocol

Topic 6, Lesson 3
Locking – a solution to concurrency

Concurrency control techniques

Two basic concurrency control techniques:

- Locking
- Timestamping

Both are conservative approaches, since they delay transactions in case they conflict with other transactions.

Optimistic methods assume a conflict is rare and only check for conflicts at commit.

Locking

Transaction uses a lock to deny access to a specific object to other transactions.

- Most widely used approach to ensure serializability.
- Generally, a transaction must claim a **shared** (**read**) or **exclusive** (**write**) lock on a data item before read or write of a data object.
- Lock prevents another transaction from modifying item or even reading it, in the case of a write lock.

Basic rules for locking objects

If transaction has a shared lock on item, can read but not update item.

If transaction has an exclusive lock on item, the transaction with the lock can both read and update item.

Reads cannot conflict, so more than one transaction can hold shared locks simultaneously on same item.

Exclusive lock gives transaction exclusive access to that item.

Some systems allow transaction to upgrade read lock to an exclusive lock, or downgrade exclusive lock to a shared lock.

Introduce method to lock an item

Write_lock(T_i, A) –request for an exclusive lock on A
Read_lock(T_i, A) – request for ask for a shared lock on A
Unlock(T_i, A) – deallocate the lock on A

If a transaction request a lock and it cannot be granted the transaction must wait for the lock to be granted

What happens if we pair the locks with the read and write of the object?

Example: schedule with an update issue

- **For two transactions, a valid schedule using these rules is:**

**S = {write_lock(T_9, bal_x), read(T_9, bal_x), write(T_9, bal_x), unlock(T_9, bal_x),
write_lock(T_{10}, bal_x), read(T_{10}, bal_x), write(T_{10}, bal_x), unlock(T_{10}, bal_x),
write_lock(T_{10}, bal_y), read(T_{10}, bal_y), write(T_{10}, bal_y), unlock(T_{10}, bal_y), commit(T_{10}),
write_lock(T_9, bal_y), read(T_9, bal_y), write(T_9, bal_y), unlock(T_9, bal_y), commit(T_9)**

Time	T_9	T_{10}
t_1	begin_transaction	
t_2	read(bal_x)	
t_3	$bal_x = bal_x + 100$	
t_4	write(bal_x)	begin_transaction
t_5		read(bal_x)
t_6		$bal_x = bal_x$ *1.1
t_7		write(bal_x)
t_8		read(bal_y)
t_9		$bal_y = bal_y$ *1.1
t_{10}		write(bal_y)
t_{11}	read(bal_y)	commit
t_{12}	$bal_y = bal_y - 100$	
t_{13}	write(bal_y)	
t_{14}	commit	

Example: incorrect schedule (2)

- If at start, $bal_x = 100$, $bal_y = 400$, result should be:
  - $bal_x = 220$, $bal_y = 330$, if T_9 executes before T_{10}, or
  - $bal_x = 210$, $bal_y = 340$, if T_{10} executes before T_9.
- However, result gives $bal_x = 220$ & $bal_y = 340$.
- S is not a serializable schedule.

Issues with the locking protocol

Creating a lock immediately before and after the database operation is not a complete solution.

Problem is that transactions may **release a lock too soon**, resulting in loss of total isolation and atomicity.

To guarantee serializability, we need an additional protocol (beyond locking) concerning the **positioning of lock and unlock operations** in every transaction.

Two-phase locking protocol

Two phases within a transaction:

- Growing phase - acquires all locks but cannot release any locks.
- Shrinking phase - releases locks but cannot acquire any new locks.

Transaction follows 2PL protocol if all locking operations precede first unlock operation in the transaction.

Preventing lost update problem with 2PL

2-phase locking force a transaction to wait

Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	write_lock(bal_x)	100
t_3	write_lock(bal_x)	read(bal_x)	100
t_4	WAIT	$bal_x = bal_x + 100$	100
t_5	WAIT	write(bal_x)	200
t_6	WAIT	commit/unlock(bal_x)	200
t_7	read(bal_x)		200
t_8	$bal_x = bal_x - 10$		200
t_9	write(bal_x)		190
t_{10}	commit/unlock(bal_x)		190

Preventing uncommitted dependency

2-phase locking force a transaction to wait

Time	T_3	T_4	bal_x
t_1		begin_transaction	100
t_2		write_lock(bal_x)	100
t_3		read(bal_x)	100
t_4	begin_transaction	$bal_x = bal_x + 100$	100
t_5	write_lock(bal_x)	write(bal_x)	200
t_6	WAIT	rollback/unlock(bal_x)	100
t_7	read(bal_x)		100
t_8	$bal_x = bal_x - 10$		100
t_9	write(bal_x)		90
t_{10}	commit/unlock(bal_x)		90

Preventing inconsistent analysis problem

2PL

Time	T_5	T_6	bal_x	bal_y	bal_z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum = 0	100	50	25	0
t_3	write_lock(bal_x)		100	50	25	0
t_4	read(bal_x)	read_lock(bal_x)	100	50	25	0
t_5	$bal_x = bal_x - 10$	WAIT	100	50	25	0
t_6	write(bal_x)	WAIT	90	50	25	0
t_7	write_lock(bal_z)	WAIT	90	50	25	0
t_8	read(bal_z)	WAIT	90	50	25	0
t_9	$bal_z = bal_z + 10$	WAIT	90	50	25	0
t_{10}	write(bal_z)	WAIT	90	50	35	0
t_{11}	commit/unlock(bal_x, bal_z)	WAIT	90	50	35	0
t_{12}		read(bal_x)	90	50	35	0
t_{13}		sum = sum + bal_x	90	50	35	90
t_{14}		read_lock(bal_y)	90	50	35	90
t_{15}		read(bal_y)	90	50	35	90
t_{16}		sum = sum + bal_y	90	50	35	140
t_{17}		read_lock(bal_z)	90	50	35	140
t_{18}		read(bal_z)	90	50	35	140
t_{19}		sum = sum + bal_z	90	50	35	175
t_{20}		commit/unlock(bal_x, bal_y, bal_z)	90	50	35	175

Cascading rollback

If **every** transaction in a schedule follows 2PL, schedule is conflict serializable.

However, problems can occur with interpretation of **when locks can be released.** We may need to hold on to the locks longer to produce a recoverable schedule.

Example: schedule with cascading rollback

2PL does not solve the problem

Time	T_{14}	T_{15}	T_{16}
t_1	begin_transaction		
t_2	write_lock($\mathbf{bal_x}$)		
t_3	read($\mathbf{bal_x}$)		
t_4	read_lock($\mathbf{bal_y}$)		
t_5	read($\mathbf{bal_y}$)		
t_6	$\mathbf{bal_x} = \mathbf{bal_y} + \mathbf{bal_x}$		
t_7	write($\mathbf{bal_x}$)		
t_8	unlock($\mathbf{bal_x}$)	begin_transaction	
t_9	⋮	write_lock($\mathbf{bal_x}$)	
t_{10}	⋮	read($\mathbf{bal_x}$)	
t_{11}	⋮	$\mathbf{bal_x} = \mathbf{bal_x} + 100$	
t_{12}	⋮	write($\mathbf{bal_x}$)	
t_{13}	⋮	unlock($\mathbf{bal_x}$)	
t_{14}	⋮	⋮	
t_{15}	rollback	⋮	
t_{16}		⋮	begin_transaction
t_{17}		⋮	read_lock($\mathbf{bal_x}$)
t_{18}		rollback	⋮
t_{19}			rollback

Cascading rollback

Transactions conform to 2PL.

T_{14} aborts.

Since T_{15} is dependent on T_{14}, T_{15} must also be rolled back. Since T_{16} is dependent on T_{15}, it too must be rolled back.

This is called **cascading rollback**.

To prevent this with 2PL, leave release of **all** locks until end of transaction. This variation of 2PL is known as **rigorous two-phase locking. Strict two-phase locking** holds only the exclusive locks until the end of the transaction.

Summary

We introduce locks to prevent 2 transactions from accessing the same data and 1 transaction intends to modify the object.

It is not enough to add locks on objects for concurrency, we also need a protocol for determining when it is “safe” to acquire a new lock and when it is ”safe” to release a lock.

The 2-phase locking protocol guaranteed serializability but not recoverability. To ensure recoverability all locks must be held until the end of the transaction (rigorous 2PL)