Discussion 06/08

Discussion 14-6

Explain the distinction between the terms *serial schedule* and *serializable schedule*.

Serial 스케쥴은 트랜잭션이 순서대로 실행되는 것. Serializable 은 병렬으로 실행되는데 그 실행 결과가 serial 스케쥴과 같으면 serializable 이라고 한다.

Discussion 14-7

Give an example of a *serializable schedule* with two transactions such that the order in which the transactions commit is different from the serialization order.

T2 - read A, write A => T1 - read A, commit => T2 - commit

이러면 T1 에서 먼저 커밋이 되었지만 serial 하게 바꾸면 T2 커밋이 먼저 되어야 한다.

Discussion 14-8

What are the values of *A* and *B* after the execution of each of these schedules, with *A*=*B*=100 initially.

The two schedules are both serializable but yield different results. Is this a discrepancy?

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

Schedule 1

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B + temp$
	write(B)
read(B)	
$B := B + 50$	
write(B)	

Schedule 2

S1 은 T1 실행 후 A 50 =〉 T2 실행 후 A 45, temp 는 5 =〉 T1 실행 후 B 150 =〉 T2 실행 후 B 155

최종적으로 A 45, B 155.

S2 는 T2 실행 후 A 90, temp 는 10 =〉 T1 실행 후 A 40 =〉 T2 실행 후 110 =〉 T1 실행 후 160

최종적으로 A 40, B 160.

=〉 교수님 설명: T1 과 T2 는 독립적인 트랜잭션. 각각은 valid. 그래서 어떤 걸 먼저 수행하는지는 DBMS 의 컨트롤 영역 밖. 따라서 둘 다 valid.

Discussion 14-9

Draw a *precedence graph* for the schedule shown on the right, and determine whether it is serializable.

T_1	T_2	T_3	T_4	T_5
read(Y) read(Z)	read(X)			
				read(V) read(W) read(W)
	read(Y) write(Y)	write(Z)		
read(U)			read(Y) write(Y) read(Z) write(Z)	
read(U) write(U)				

T1 => T2, T3, T4

T2 => T4

T3 => T4

사이클이 없기 때문에 serializable 함.

Discussion 14-10

Define *recoverable schedule*.

Is the following schedule recoverable?

When is a schedule recoverable?

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)
	commit
abort	

해보기!

Discussion 14-11

Define *cascadeless schedule*.

Is the following schedule cascadeless?

When is a schedule cascadeless?

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)
abort	

이것도 해보기!

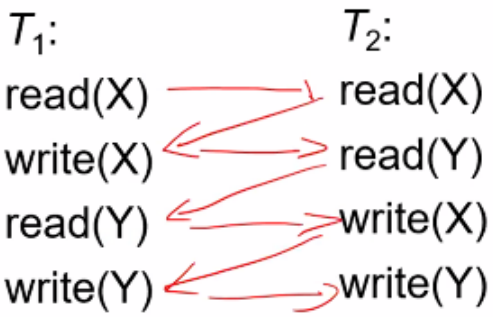
Discussion 14-12

Show that *every cascadeless schedule is also recoverable.*

Cascadeless 는 한 트랜잭션에서 write 를 하고 commit 한 뒤에 다른 트랜잭션에서 read 를 한다. Recoverable 의 정의가 write, read 가 있을 때 write 하는 트랜잭션의 커밋이 read 하는 트랜잭션의 커밋보다 빨라야 하는 것이므로, cascadeless 에서는 write 한 뒤 커밋을 하지 않으면 다른 트랜잭션에서 읽을 수가 없으므로 항상 성립한다.

Discussion 15-1

Execute the two transactions by alternating the instructions observing the *two phase locking protocol* (*2PL*). Whenever there is a *deadlock*, abort the transaction of the last lock request and restart it. Start with T_1.



~~T1 read(X) => T2 read (X) => T1 write(X). 여기서 T1 에서 X 에 대해 lock-X 획득 => T2 read(Y) => T1 read(Y) => T2 에서 write(X)를 할 때 deadlock 발생.~~

이거 틀림!

=> T1 에서 write(X)를 할 때 T2 도 X 에 대해 lock-S 를 들고 있으므로 기다려야 함. 그래서 T2 에서 read(Y)를 하고 write(X)를 할 때는 T1 에서 lock-S 있으므로 데드락 발생.

그래서 T2 를 롤백함. => T1 이 write(X) 수행. 그 뒤 T2 에서 T1 이 X 에 대해 lock-X 를 들고 있으므로 read(X)를 할 때 기다림. => T1 이 read(Y) 수행. T2 는 여전히 기다림. => T1 이 write(Y)까지 수행. => T1 이 모든 lock 해제. => T2 에서 read(X)부터 write(Y)까지 모두 수행.

Discussion 15-2

Explain and compare *strict* 2PL and *rigorous* 2PL.

Strict 2PL 은 exclusive lock 을 commit 이나 abort 할 때까지 release 하면 안 되는 규칙.

Rigorous 2PL 은 shared lock 까지 포함.

Discussion 15-3

Both *strict* 2PL and *rigorous* 2PL ensure conflict serializability and cascadelessness (thus, recoverability) while both are not free from deadlocks.

Since rigorous 2PL holds both shared locks and exclusive locks until the transaction commits, it allows less concurrency than strict 2PL.

Then why do DBMSs implement rigorous 2PL at all?

Rigorous 2PL 은 모든 락을 획득하는 순간 release 할 수 없으므로 트랜잭션을 serialize 하기 쉬움.
트랜잭션들이 커밋된 순서대로 serialize 하면 됨.

=〉 트랜잭션 매니저가 growing phase 인지 shrinking phase 인지 판단하기가 거의 불가능에 가까움.
그래서 구현을 편하게 하기 위해 rigorous 2PL 을 많이 씀.

Discussion 15-4

The *lock manager* is a module within a DBMS that handles lock requests. How would you implement a lock manager?

1) data structure
2) APIs

해보기!