Chapter 15 : 동시성 제어

Database System Concepts, 6th Ed.

동시성 제어 (Concurrency Control)

- 데이터베이스 시스템은 모든 동시 스케쥴이 다음 조건을 만족시키도록 관리하여야 한다.
  - 충돌 혹은 뷰 직렬 가능
  - 회복 가능하며, 가능하다면 연쇄 복귀 없는 스케쥴

- 스케쥴이 실행된 후 직렬성을 검사하는 것은 너무 늦다.

- 목표 — 직렬성을 보장할 동시성 제어 규약을 개발하는 것.
  - 규약은 일반적으로 비직렬 가능 스케쥴을 회피하는 원칙을 부과한다.

Chapter 15-1:동시성 제어

- 로크 기반 규약
- 교착상태 처리
- 타임스탬프 기반 규약

로크 기반 규약 (Lock-Based Protocols)

- 로크는 데이터 항목에 대한 동시 액세스를 제어하는 기법이다.

- 데이터 항목은 두기지 유형으로 로크할 수 있다.

  1. 배타(X)유형(*exclusive mode*). 데이터 항목을 읽고 쓸 수 있다.
     배타 로크는 **lock - X** 명령으로 요청한다.
  2. 공유(S)유형 (*shared mode*). 데이터 항목을 읽을 수만 있다.
     공유 로크는 **lock - S** 명령으로 요청한다.

- 로크 요청은 동시성 제어 매니저에게 요청하며, 요청이 허용된 후에 만 트랜잭션이 진행될 수 있다.

로크 기반 규약(계속)

- 로크 양립성 행렬 (**Lock-compatibility matrix**)

	S	X
S	true	false
X	false	false

- 트랜잭션에서 어떤 데이터 항목에 로크를 요청할 때, 다른 트랜잭션들이 이미 걸고 있는 로크와 양립할 수 있을 때만 허용된다.
- 위의 행렬에서는 몇 개의 트랜잭션이라도 어떤 데이터 항목에 공유 로크를 걸 수 있다.
  - 그러나, 어떤 트랜잭션이 데이터 항목에 배타 로크를 걸고 있으면 다른 트랜잭션에서는 어떠한 로크도 걸 수 없음을 보이고 있다.
- 로크가 허용되지 않으면, 요청한 트랜잭션은 다른 트랜잭션들이 걸고 있는 모든 양립할 수 없는 로크가 해제될 때까지 기다려야 한다. 그 이후에 로크가 허용된다.

로크 기반 규약(계속)

- 로킹 규약은 로크를 요청하고 해제하는 동안 모든 트랜잭션이 따라야 할 규칙의 집합이다.
- 로킹 규약은 가능한 트랜잭션들의 실행 스케쥴의 수를 제한한다.
- 로킹을 수행하고 있는 트랜잭션의 예:

T1:

lock-X(B);
read(B);
B:=B-50;
write(B);
unlock(B);
lock-X(A);
read(A);
A:=A+50;
write(A);
unlock(A).

T2:

lock-S*(A);*
read *(A)*;
unlock*(A)*;
lock-S*(B)*;
read *(B)*;
unlock*(B)*;
display*(A+B).*

- 위와 같은 로킹은 직렬성을 보장하기에는 불충분하다 - A 와 B의 read 사이에 A 와 B 가 갱신되면 출력되는 합계가 틀린다.

로크 기반 규약(계속)

T1:

lock-X(B);
read(B);
B:=B-50;
write(B);
unlock(B);
lock-X(A);
read(A);
A:=A+50;
write(A);
unlock(A).

T2:

lock-S*(A);*
read *(A);*
unlock*(A);*
lock-S*(B);*
read *(B);*
unlock*(B);*
display*(A+B).*

T1	T2
1. **read**(*B*)	
2. *B* := B – 50	
3. **write**(B)	
	1. read(A), read(B), print(A+B)
4. **read**(A)	
5. *A* := A + 50	
6. **write**(A)	

스케쥴 1

T1	T2	Concurrency-control manager
lock-X(B)		
		grant-X(B, T1)
read(B)		
B:=B-50		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T2)
	read(B)	
	unlock(B)	
	display(A+B)	
lock-X(A)		
		grant-X(A, T1)
read(A)		
A:=A-50		
write(A)		
unlock(A)		

- 위와 같은 로킹은 직렬성을 보장하기에는 불충분하다 (스케쥴 1)
  A 와 B의 read 사이에 A 와 B 가 갱신되면 출력되는 합계가 틀린다.

로크 기반 규약(계속)

T1, T2의 로크를 해제하는 것을 트랜잭션의 마지막에 수행하는 것으로 변경

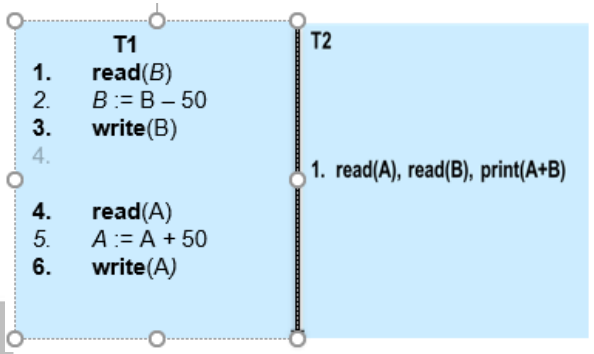
T3:

lock-X(B);
read(B);
B:=B-50;
write(B);
lock-X(A);
read(A);
A:=A+50;
write(A);
unlock(B);
unlock(A).

T4:

lock-S*(A);*
read *(A)*;
lock-S*(B)*;
read *(B)*;
display*(A+B).*
unlock*(A)*;
unlock*(B)*;

로크 기반 규약의 함정



- 다음과 같은 부분 스케쥴을 고려해 보자

스케쥴 2

T3	T4
lock-X(B);	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).	

T_3	T_4
lock-x (B)	
read (B)	
$B := B - 50$	
write (B)	
	lock-s (A)
	read (A)
	lock-s (B)
lock-x (A)	

- T_3 도 T_4도 처리를 진행할 수 없다- **lock-s(B)**를 실행하면 T_3 가 B에 걸고 있는 로크를 해제할 때까지 T_4가 기다리게 되고, 반면에 **lock-X(A)**를 실행하면 T_4가 A에 걸고 있는 로크를 해제할 때 까지 T_3 가 기다리게 된다.
- 그러한 상태를 **교착상태**라 한다. 교착상태를 처리하려면 T_3 또는 T_4 중 하나를 복귀시켜 해당 로크가 해제되도록 해야 한다.

로크 기반 규약의 함정(계속)

- 교착상태의 가능성은 대부분의 로킹규약에 존재한다. 교착상태는 필요악이다.
- 또한 동시성 제어 매니저가 잘못 설계되면 **기아(starvation)** 의 가능성이 있다. 예를 들어:
  - 일련의 트랜잭션들이 같은 데이터 항목에 공유 로크를 요청하고 허용되고 있는 동안, 어떤 트랜잭션은 그 데이터 항목에 배타 로크를 기다리고 있다.
  - 동일한 트랜잭션이 교착상태로 인해 반복해서 복귀된다.
- 동시성 제어 매니저는 기아를 방지하도록 설계될 수 있다.

2 단계 로킹 규약(Two-Phase Locking Protocol)

- 충돌 직렬 가능 스케쥴을 보장하는 규약이다.
- 단계 1: 증대 단계 (Growing Phase)
  - 트랜잭션은 로크를 얻을 수 있다.
  - 트랜잭션은 로크를 해제할 수 없다.
- 단계 2: 수축 단계 (Shrinking Phase)
  - 트랜잭션은 로크를 해제할 수 있다.
  - 트랜잭션은 로크를 얻을 수 없다.
- 이 규약은 직렬성을 보장한다. 트랜잭션들은 그들의 로크 시점 (트랜잭션이 마지막 로크를 얻는 시점) 순서로 직렬화될 수 있음을 증명할 수 있다.

2 단계 로킹 규약(계속)

- 2 단계 로킹은 교착상태를 피하는 것을 보장하지는 못한다.
- 2 단계 로킹 하에서 연쇄복귀가 발생 가능하다. (T_5이 실패(failure)하면 T_6과 T_7 또한 복귀(rollback) 되어야 한다.

T_5	T_6	T_7
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)

abort

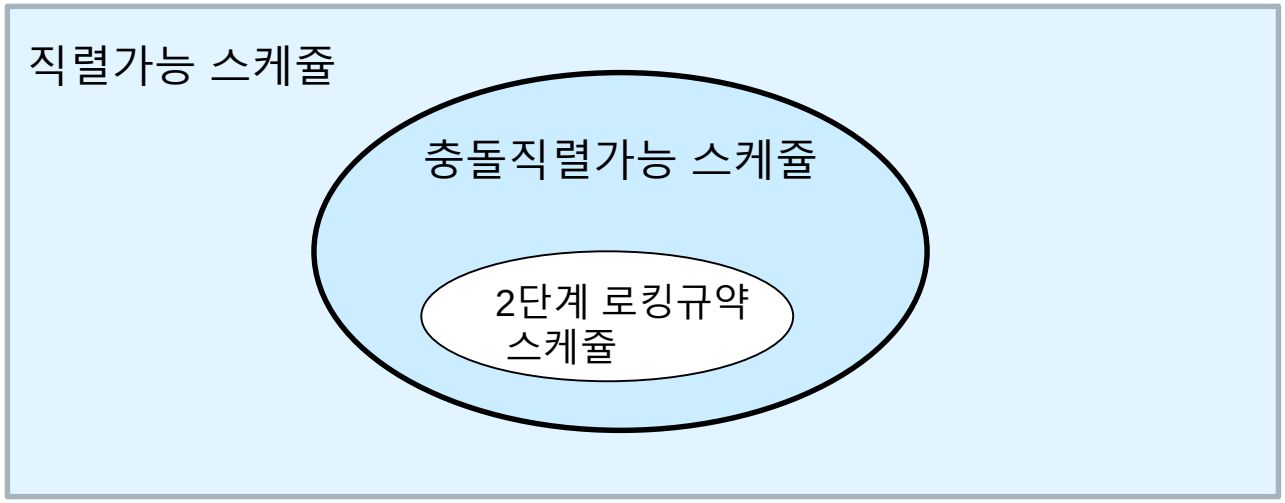
- 이를 피하려면(연쇄복귀가 발생하지 않게 하려면, 정확한 2 단계 로킹(**strict two-phase locking**)이라는 수정된 규약을 따른다. 이 규약에서 트랜잭션은 완료 또는 중단할 때까지 모든 배타 로크를 유지하여야 한다.
- 엄격한 2 단계 로킹 (**Rigorous two-phase locking**)은 보다 더 엄하다 : 이 규약에서는 모든 로크가 완료 또는 중단할 때까지 유지된다. 이 규약에서는 트랜잭션들이 완료하는 순서대로 직렬화할 수 있다.

2 단계 로킹 규약(계속)

- 2 단계 로킹을 사용하면 얻을 수 없는 충돌 직렬 가능 스케쥴이 있을 수 있다.

- 그러나, 별도의 정보가 없으면(예를 들어, 데이터 액세스 순서), 다음과 같은 의미에서 충돌 직렬성을 위해 2 단계 로킹이 필요하다:

  2 단계 로킹을 따르지 않는 트랜잭션 T_i가 주어지면, 2 단계 로킹을 따르는 트랜잭션 T_j와 충돌 직렬 가능하지 않은 T_j 와 T_i의 스케쥴이 존재할 수 있다.



로크변환

다음의 두 트랜잭션을 살펴보자.

T8:

read(a1);
read(a2);
…
read(an);
write(a1).

T9:

read(a1);
read(a2);
display(a1+a2).

lock-X(a1)
lock-S(a2) …
read(a1);
read(a2);
…
read(an);
write(a1);
unlock(a1)
unclock(a2) …

lock-S(a1)
lock-S(a2)
read(a1);
read(a2);
display(a1+a2)
unlock(a1)
unclock(a2)

lock-S(a1)
lock-S(a2)
read(a1);
read(a2);
display(a1+a2)
unlock(a1)
unclock(a2)

lock-X(a1)
lock-S(a2) …
read(a1);
read(a2);
…
read(an);
write(a1);
unlock(a1)
unclock(a2) …

로크 변환

- 로크 변환을 가진 2 단계 로킹: 이 규약은 직렬성을 보장한다. 그러나, 여전히 프로그래머가 다양한 로킹 명령을 삽입해야 한다.
  - 첫번째 단계:
    - 항목에 **lock-S** 를 얻을 수 있다.
    - 항목에 **lock-X**를 얻을 수 있다.
    - **lock-S**를 **lock-X**로 변환할 수 있다. (**상향조정**)
  - 두번째 단계:
    - **lock-S**를 해제할 수 있다.
    - **lock-X**를 해제할 수 있다.
    - **lock-X**를 **lock-S**로 변환할 수 있다. (**하향조정**)

T8

```
read(a1);
read(a2);
...
read(an);
write(a1).
```

T9

```
read(a1);
read(a2);
display(a1+a2).
```

T_8	T_9
lock-s (a_1)	
	lock-s (a_1)
lock-s (a_2)	
	lock-s (a_2)
lock-s (a_3)	
lock-s (a_4)	
	unlock-s (a_3)
	unlock-s (a_4)
lock-s (a_n)	
upgrade (a_1)	

자동 로크 획득

- 트랜잭션 T_i는 명시적인 로킹 호출없이 표준 read/write 명령을 낸다.
  - read(D) 연산은 다음과 같이 처리된다 :

```
if Ti has a lock on D
  then
        read(D)
  else begin
          if necessary wait until no other
             transaction has a lock-X on D
          grant Ti a  lock-S on D;
          read(D)
        end
```

자동 로크 획득(계속)

- write(*D*)는 다음과 같이 처리된다 :

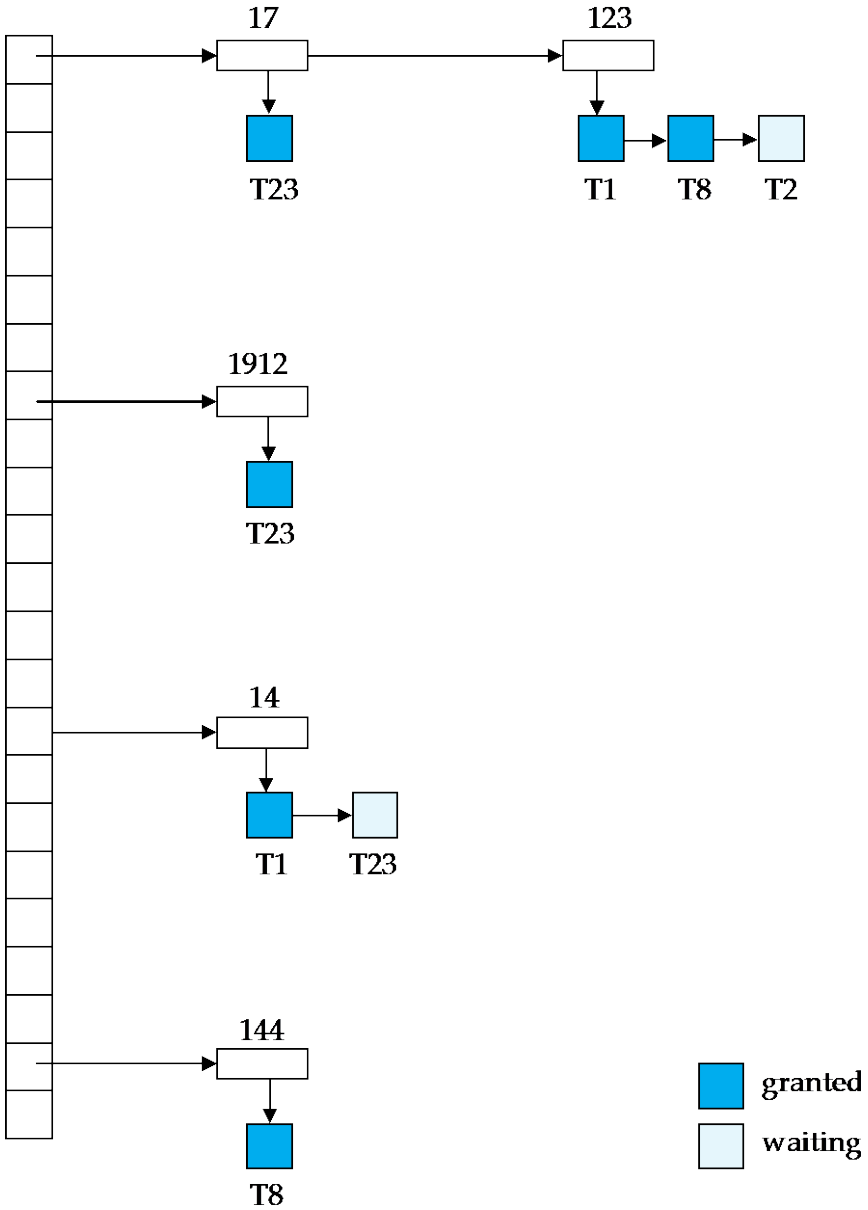
```
if Ti has a lock-X on D
  then
    write(D)
  else begin
      if necessary wait until no other trans. has any lock on D,
      if Ti has a lock-S on D
          then
              upgrade lock on D to lock-X
          else
              grant Ti a lock-X on D
          write(D)
    end;
```

- 모든 로크는 완료 또는 중단 후 해제된다.

로킹 구현 방법

- 트랜잭션들이 보내는 로크 요청/해제 메시지를 처리하는 독립된 프로세서로서 로크 매니저(**lock manager**)가 구현될 수 있다.
- 로크 매니저는 로크 요청에 대하여 허용(grant) 메시지를 보낸다 (혹은 deadlock 발생시 트랜잭션에게 roll back을 요청한다)
- 로크를 요청한 트랜잭션은 요청에 대한 답이 올 때까지 기다린다.
- 로크 매니저는 로크 테이블(**lock table**)을 관리한다.
- 로크 테이블은 일반적으로 데이터 항목명으로 인덱싱 되는 인-메모리 해쉬 테이블 구조로 구현된다.

로크 테이블



- Black rectangle : granted locks

  white rectangle: waiting requests
- 로크 테이블은 로크 타입(granted or requested)을 기록한다.
- 새로운 요청이 들어오면 해당 데이터 아이템의 요청 큐의 마지막에 요청 항목이 더해지고, 해당 요청이 기존 로크들과 양립성을 만족시키는 경우에는 요청이 허용된다.
- 로크 해제(unlock) 요청이 있으면 해당 로크를 삭제하고, 요청 큐에 쌓여있는 이 후의 요청들이 허용될 수 있는지 검사한다.
- 만약 트랜잭션이 중단되면, 그 트랜잭션의 모든 (기다리고 있거나 허용 되었거나에 상관없이) 요청은 삭제된다
  - 로크 매니저는 효율성을 위하여 각 트랜잭션에서 사용된 모든 로크 리스트를 유지 관리할 수 있다.

Thank You