24. Concurrency Control

▼ Lock-Base Protocols

lock → concurrent한 접근을 control

- exclusive(X) mode : 하나만 접근 가능 → read, write 모두 사용 가능
- shared(S) mode : 한 번에 여러 개 접근 가능 → read만 사용 가능

lock request는 grant 받은 후여야 Transaction이 실행 가능

- Lock-compatiablity matrix

	S	X
S	true	false
X	false	false

Shared는 여러 transaction 가능!!.

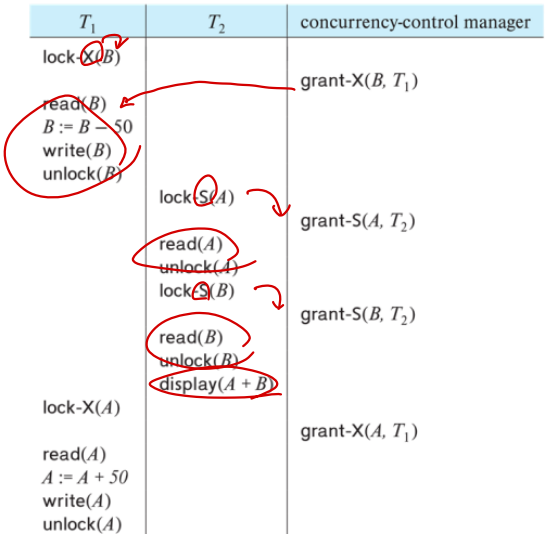
- example

```
T2: lock-S(A);
    read (A);
    unlock(A);
    lock-S(B);
    read (B);
    unlock(B);
    display(A+B)
```

→ serializability가 guarantee되지 않음

(unlock 하는 동안 다른 transaction이 lock 가져갈 수 있음.)

- schedule with Lock Grants → serializability 보장되지 않는 version

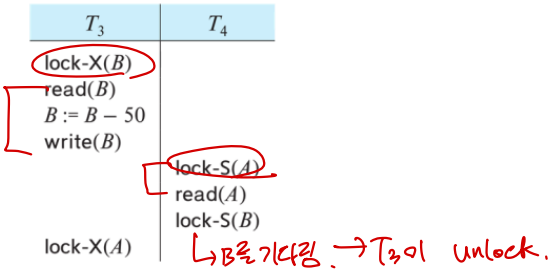


⇒ serializabilty가 보장되지 않음

→ requsting과 releasing하는 것의 순서가 보장되지 않음

→ 시작할 때 lock을 다 가지고 시작해야 함.

▼ Deadlock



⇒ deadlock 걸리게 됨(서로가 서로의 lock을 기다림)

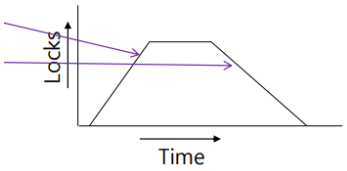
→ T_3 혹은 T_4가 rollback + release 되어야 함

- starvation
  - 다른 transaction이 lock에 대해 계속 굶주림
  - deadlock에 의해 동일한 transaction이 계속 반복적으로 roll back됨

⇒ concurrency control manager : starvation을 막도록 designed 될 수 있음

▼ 1. Two-Phase Locking protocol

serializability 보장함 → 시작할 때 lock을 다 가지고 시작



- phase1 : Growing Phase
  - transaction → lock을 얻는 구간
- phase2 : Shrinking Phase
  - transaction → unlock하는 구간(lock을 얻으려 하면 안 됨)

- conflict-serializability가 보장됨
  - conflict-serializable schedule을 위해 꼭 필수적으로 존재하는 것은 아님
  - 정렬된 순서로 실행될 수 있음
- recoverability를 보장하기 위해서는?
  - String 2PL : exclusive lock을 commit/abort 하기 전까지 가지고 있음
  - Rigorous 2PL : 모든 lock을 commit/abort 하기 전까지 가지고 있음

    ⇒ recoverability 보장 + cascading roll-backs

- 하지만, deadlock 방지는 불가함!

- with lock conversions
  - Growing Phase lock을 걺
    - lock-S, lock-X 가질 수 있음
    - + lock-S에서 lock-X로 convert 가능
  - Shrinking Phase lock을 풂
    - lock-S, lock-X release 가능
    - + lock-X에서 lock-S로 convert 가능

  ⇒ serializability 보장

  - Automatic Acquisition of Lock

    Ti → standard r/w instruction, without explicit locking calls
    - read()

```
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
```

- write()

```
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;
```

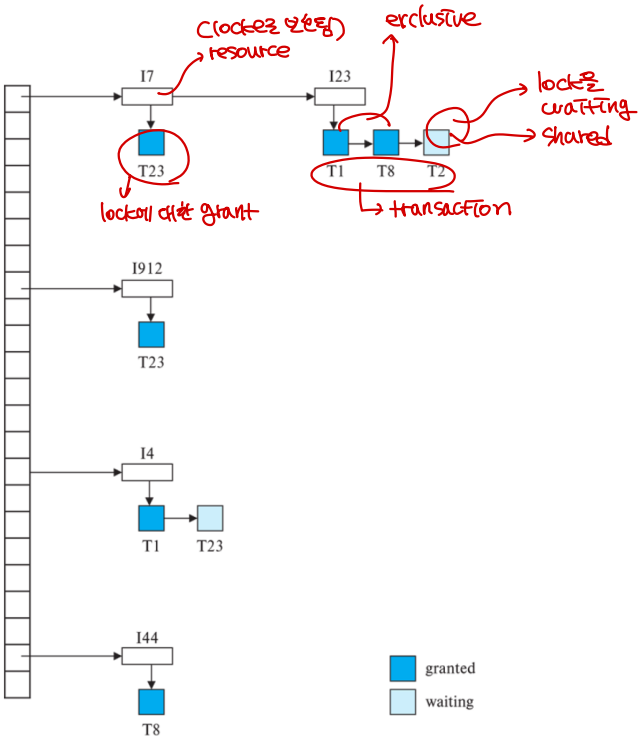
▼ implementation of locking

전통적인 DB에 해당되는 사항

- lock manager를 독립적인 process로 여겨보기
- transaction → lock manager : lock, unlock 요청을 메세지처럼 보냄
- lock manager → transaction : lock 허용 혹은 deadlock일 때 lock 비허용을 메세지로 보냄

⇒ lock table 사용 : lock granted 혹은 pending reques한 것을 저장

- Lock Table



- transaction이 abort → 기다리거나 lock을 받은 것들 모두 지워짐
- lock manager : 각 transacton에 대해 가지고 있는 lock list 저장 → 효율!

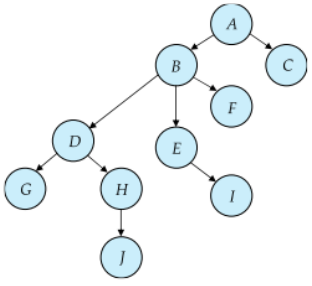
▼ 2. Graph Based Protocols

2PL 대신 사용함

- data item set D = {d1, d2, ... , dn} : locking 순서를 정해둠

  → D는 database graph임(directed acyclic graph)
- transaction이 di → dj 일 때 반드시 di를 접근한 뒤 dj를 접근해야 함

- tree protocol



- 오직 exclusive lock만 allowed 됨
- unlock은 언제나 가능
- 첫번째 거는 lock은 무엇이든지 다 걸 수 있음

- 두번째 이후로는 parent에 lock을 걸어야 lock을 걸 수 있음
- 한 번 unlock한 transaction은 동일 data item에 대해 relock 불가

- 장점
  - conflict serializability 보장
  - deadlock free 보장 → data item 항상 정해진 순서에 따라 hold
  - 2PL보다 unlocking이 일찍 일어남

    → waiting time 줄임

    ⇒ concurrency 증가
- 단점
  - protocol → recoverability 혹은 cascade freedom 보장 x
    - commit dependency가 필요함
  - transaction → 접근하지 않는 data item에 대한 lock 필요할지도 모름
    - locking overhead가 커지고 추가적인 waiting time 발생 가능

      ⇒ concurrency가 잠재적으로 감소함

⇒ schedule이 어느정도 가능함

▼ Deadlock handling

- Deadlock prevention
  - system이 절대 deadlock state에 들어가지 않도록 보장
  - 가장 기본적인 방법
    1. pre-declaration : 각 transition이 실행을 시작하기 전에 data item에 대해 모두 lock 걸음
    2. graph-based protocol : 모든 data item에 대해 미리 순서 정함 + 이 순서대로 lock 부여

  - preemptive 관점
    - 모든 transaction → 실행 시점에 system으로부터 timestamp 받고 transaction 분류함

    1. wait-die(non-preemptive)

- older : younger가 가지고 있는 lock 얻기 위해 younger의 release 기다림
- younger : older가 가지고 있는 lock이 필요하다면 기다리지 않고 roll back → 자신이 가지고 있는 lock release

⇒ but, transaction이 lock을 얻기 전에 여러 번 die할 수 있음

2. wound-wait
   - older : younger를 기다리지 x, 걍 younger roll back해서 뺏어옴
   - younger : older를 기다림

   ⇒ wait-die보다 roll back 적음

- roll back transition : 기존 timestamp로 restart
  - older가 newer보다 우선 → starvation 방지

- Timeout-Baed Schemas
  - 각 transaction은 특정 시간만큼 lock을 기다림

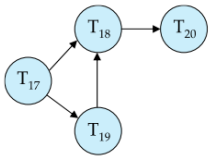
    → 이후 wait time out + roll back
  - 간단한 구현으로 deadlock 피할 수 있음 → 대부분의 DBMS에서 채택
  - deadlock이 발생하더라도 일정 시간이 지나면 transaction roll back됨

    → deadlock 해소

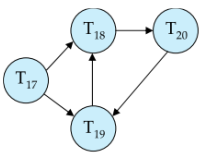
  - 단점
    - timeout 간격 정하기 어려움
    - starvation 발생 가능
    - 불필요한 rollback 발생 가능

▼ Deadlock Detection

- Wait-for graph



Wait-for graph without a cycle



Wait-for graph with a cycle

- verticies : transactions
- edge Ti→Tj : Tj에 의해 conflict mode에서 lock을 기다리는 Ti

- deadlock state ↔ cycle 만드는지 검사

▼ Deadlock Recovery

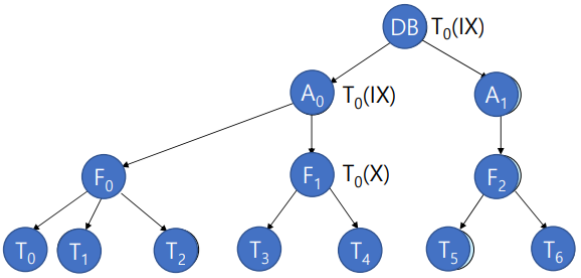
- deadlock이 detect되었을 때
  - 몇몇 transaction(victim)을 rollback시켜 deadlock 해소
    - victim : roll back으로 낭비가 가장 적은 transaction 선택
  - victim
    - transaction 전체 roll back
    - 다시 시작 혹은 lock을 놓기 위해 필요한 부분만 roll back 가능

but, starvation 발생 가능.. 비용 큼..

▼ Multiple Granularity

lock의 단위를 다르게 가져가는 기법 → tree처럼 표기

- root가 가장 큰 단위
  - database → area → file → record



- Fine Granularity(lower in tree) : high concurrency, high locking overhead
- Coarse Granularity(higher in tree) : low ''
- Multiple Granularity : Find, Coarse 짬뽕

- Intention lock

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

    - intention-shared(IS) : 미래의 하위 노드에도 shared lock 얻음
    - intention-exclusive(IX) : 미래의 하위 노드에도 exclusive lock 얻음
    - shared and intention-exclusive(SIX) : 미래의 하위 노드에 다 얻음

- lock granarity escalation
    - 특정 level에 너무 많은 lock이 있다면
    - S와 X lock을 high granularity로 바꿀 수 있음

▼ Date insert / write

- locking rule
    - insert, delet 모두 X-lock 얻어야 가능
- 무엇을 보장?
    - r/w가 delete와 conflict
    - 삽입 transaction이 commit될 때까지 다른 transaction에서 inserted tuple에 access할 수 없음
- phantom phenomenon(유령 현상)
    - example
        - T1 transaction이 relation에 대해 predicate read을 하려 함

```
select count(*)
 from instructor
 where dept_name = 'Physics'
```

        - T2 transaction이 predicate read가 끝나기 전에 T1이 active인 동안 insert tuple

```
insert into instructor values ('11111', 'Feynman', 'Physics', 94000)
```

        ⇒ conflict

    - 만약 tuple lock만 사용된다면 serializable schedule이 아닌 결과가 나올 수 있음

- example 2

T1	T2
Read(instructor where dept_name='Physics')	
	Insert Instructor in Physics
	Insert Instructor in Comp. Sci.
	Commit
Read(instructor where dept_name='Comp. Sci.')	

- handling phantoms ⇒ Data Item을 Relation에 연결하여 Relation에 포함된 tuple에 대한 정보 나타내기!
  → 예) relation scan transaction ⇒ shared lock
  insert/delete ⇒ exclusive lock
- prevent phantoms
  - simple solution
    1. relation에 data item 연결
    2. relation scan하는 transaction : shared lock 획득
    3. tuple insert/deletion하는 transaction : exclusive lock 획득
    4. data 항목 lock → 각 tuple에 대한 lock과 충돌하지 않음

    → insertion, deletion concurrency가 낮아짐
  - index locking protocol
  - next-key locking protocol