# DB 트랜잭션

고재원

발표 도중 궁금한 것이 있으면
디스코드 부부젤라를 불 것!!!!!

# 목차

- 트랜잭션
- 트랜잭션의 이해
- DBMS의 의무, ACID
- 트랜잭션의 상태
- 트랜잭션과 동시성
- 동시성에서 일어 날 수 있는 이상현상
- 격리 레벨
- 실제 DBMS의 격리 레벨

# 트랜잭션

- DB에서 논리적으로 분해가 불가능한 하나의 작업 단위

- 라고 하는데 이게 뭐지?!

# 내 계좌에서 A의 계좌로 10만원을 송금 할거야!

- 이 과정을 DBMS의 입장에서 보면, **여러 과정**으로 나눠진다.

1. 내 계좌를 db에서 읽어서 버퍼에 가져온다.
2. 내가 가진 금액에서 10만원을 뺀다.
3. 뺀 결과를 db에 저장한다.
4. A의 계좌를 db에서 읽어서 버퍼에 가져온다.
5. A가 가진 금액에서 10만원을 더한다.
6. 더한 결과를 db에 저장한다.

# 내 계좌에서 A의 계좌로 10만원을 송금 할거야!

조금 간소화 하면?

- Read(jw)
- jw -= 100,000 in buffer
- Write(jw)
- Read(A)
- A += 100,000 in buffer
- Write(A)

즉 내 계좌에서 A로 송금을 할 때
이와 같은 여러 연산들을 수행해야 한다.

즉 하나의 작업
= 여러 연산들의 합이며,

이 연산들이 순서대로 문제 없이 일어나면서 작업 하나가 실행되는 것이다.

이 작업 단위를 **트랜잭션**이라 한다.

# 그렇다면 이 작업(트랜잭션)에서 고려해야 할 것

T
- Read(jw)
- jw -= 100,000 in buffer
- Write(jw)
- Read(A)
- A += 100,000 in buffer
- Write(A)

**1. 시스템 고장이나 하드웨어 고장으로 작업 도중 실패가 나면?**

내 계좌에서 10만원 뺀 것을 write했는데 이때 고장이 나서 이후 연산이 실행이 안 된다면?

**2. 다른 트랜잭션들과 경쟁을 해야 하는 경우에는?**

재우가 카카오 정산하기를 요청해서 우리가 재우에게 10만원씩 송금을 하는 경우, 재우 계좌에 대한 여러 트랜잭션들이 발생, 어떻게 경쟁?

# 이건 누가 해? 개발자인 우리가?

ㄴㄴ. DBMS가 한다.

# 그래서 DBMS는 각 트랜잭션에 대해…

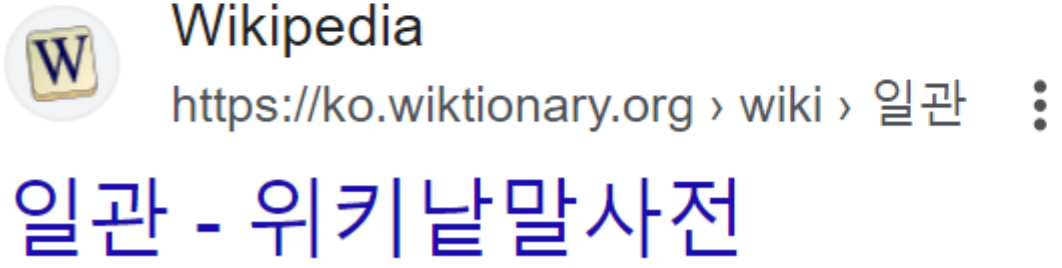
1. Atomicity, 원자성을 유지해야 한다. (아까 1번의 손실을 방지하기 위해)
- 부분적으로 실행 된 트랜잭션은 절대로 DB에 반영되지 않도록 해야 한다.

2. Durability, 지속성을 유지해야 한다.
- 모든 연산이 완료된 트랜잭션이 있다면, 이 트랜잭션은 소프트웨어/하드웨어 문제 발생에 상관없이 무조건 DB에 반영이 되어야 한다.

# 그래서 DBMS는 각 트랜잭션에 대해…

3. Consistency, 일관성을 유지해야 한다.

1. 하나의 방법이나 태도로써 처음부터 끝까지 한결같음.



# 그래서 DBMS는 각 트랜잭션에 대해…

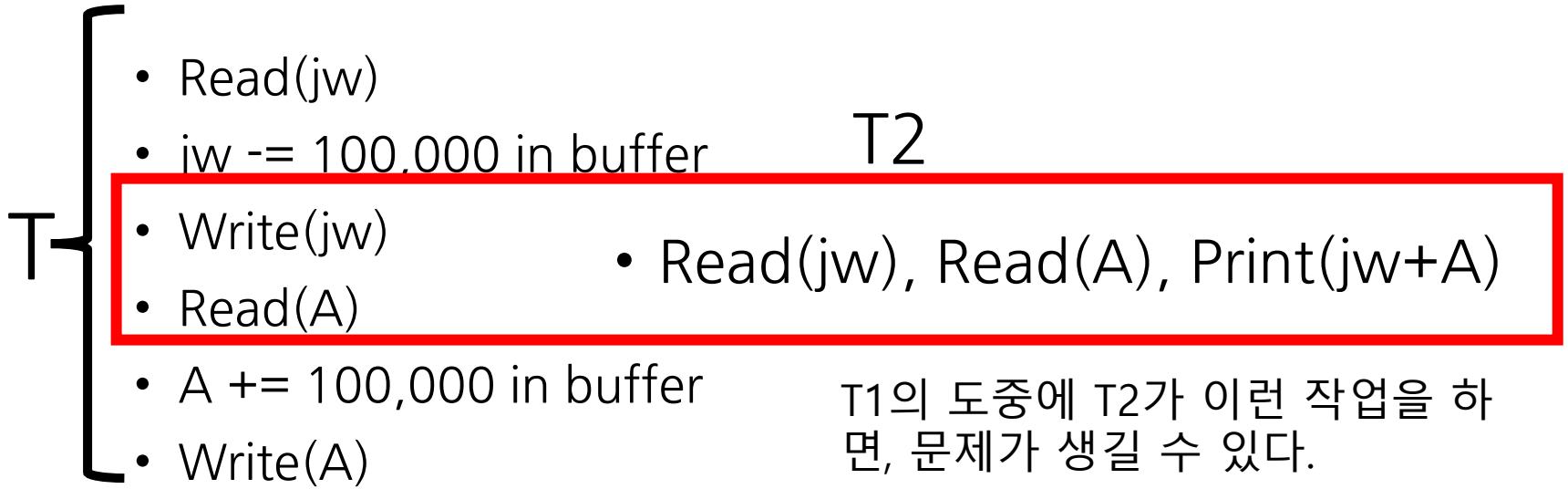
## 3. Consistency, 일관성을 유지해야 한다.

- 명시적으로 알려진 “무결성 제약 조건”, 기본 키와 외래키 등을 만족해야 한다.
- 암묵적인 무결성 제약조건, 계좌 이체 후 전체 합은 이전과 같아야 한다 등을 만족해야 한다.

- 트랜잭션은 일관적인 db를 참조해야 한다.
- 트랜잭션을 실행하는 동안 db는 일시적으로 일관적이지 않을 수 있다.
- 하지만 트랜잭션이 성공적으로 끝나면, db는 반드시 일관적이어야 한다.
- 에러 가능성이 있는 트랜잭션이 db를 일관적이지 않게 만든다. (조회 vs 업데이트 경쟁)

# 그래서 DBMS는 각 트랜잭션에 대해…

4. Isolation, 격리성을 유지해야 한다.




- 각 트랜잭션이 db를 독점하는 것 처럼 느끼도록 허상을 심어줘야 한다.
-각 트랜잭션을 순차적으로, 즉 serial 하게  실행하면 격리성을 유지 할 수 있다.
- 하지만, 여러 트랜잭션을 동시에 실행시키면 여러 장점들이 있을 수 있다.

# 무결성이 그래서 뭐임?!

나도 몰라!
발표 자료 만드느라 못 찾았어!!

# 요약하자면…. ACID

**트랜잭션**은 여러 데이터에 접근하거나 업데이트를 하는 **작업의 단위**이며,
DBMS는 데이터 무결성을 위해 다음과 같은 조건을 만족해야 한다.

## Atomicity 원자성

트랜잭션은 모든 내부 연산이 적용되거나 모든 내부 연산이 적용되지 않아야 한다.

## Consistency 일관성

독립적으로 작동하는 트랜잭션은 DB의 일관성을 유지해야 한다.

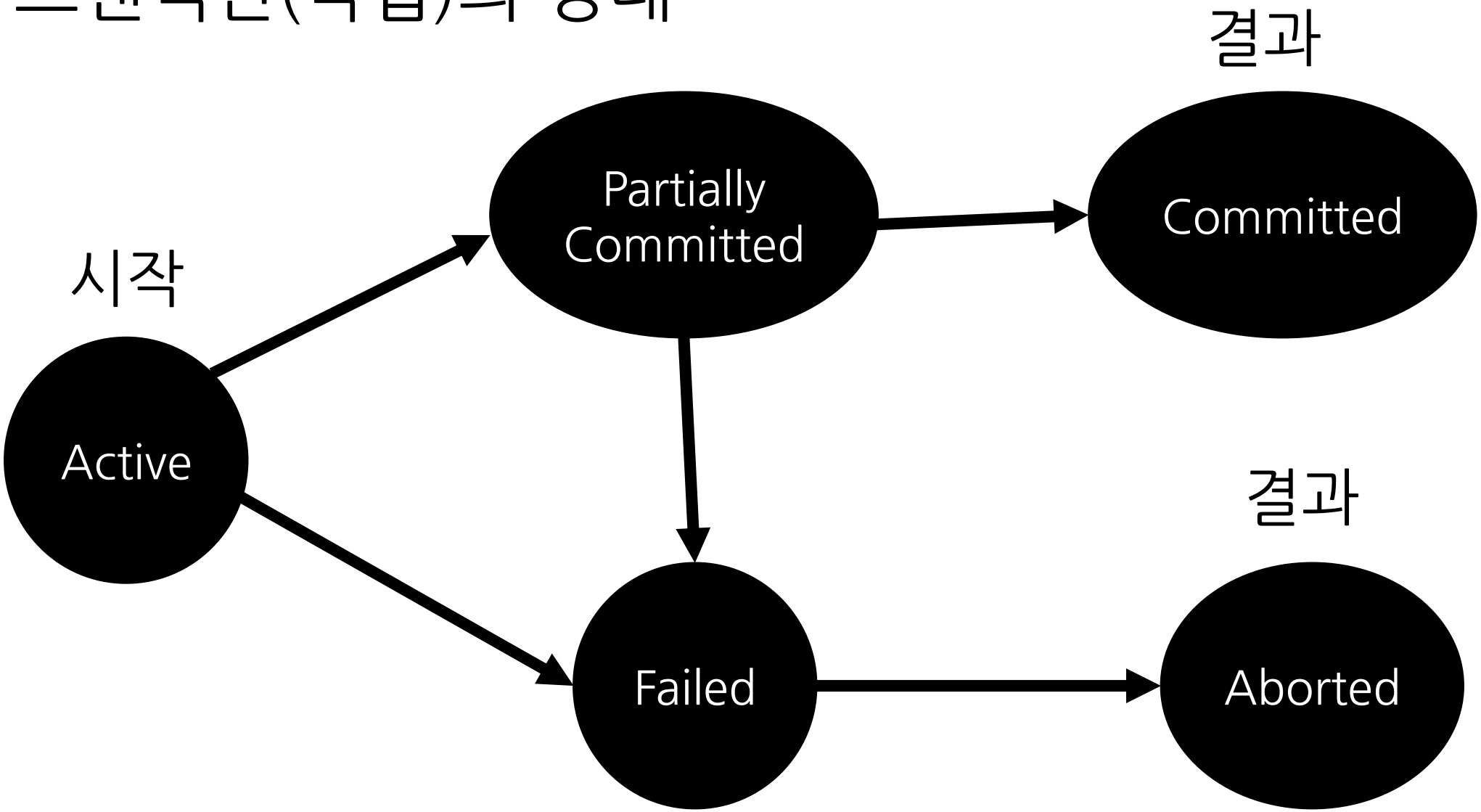
## Isolation 격리성

여러 트랜잭션이 동시에 발생 할 수 있지만, 각 트랜잭션은 동시에 작동 중인 다른 트랜잭션을 신경 쓰지 않아야(unaware of) 한다.

## Durabillity 지속성

트랜잭션이 성공적으로 끝나면, 하드웨어나 소프트웨어 문제에 상관없이 무조건 결과가 DB에 적용이 되어야 한다.

# 트랜잭션(작업)의 상태


결과
Partially Committed
Committed
시작
Active
결과
Failed
Aborted

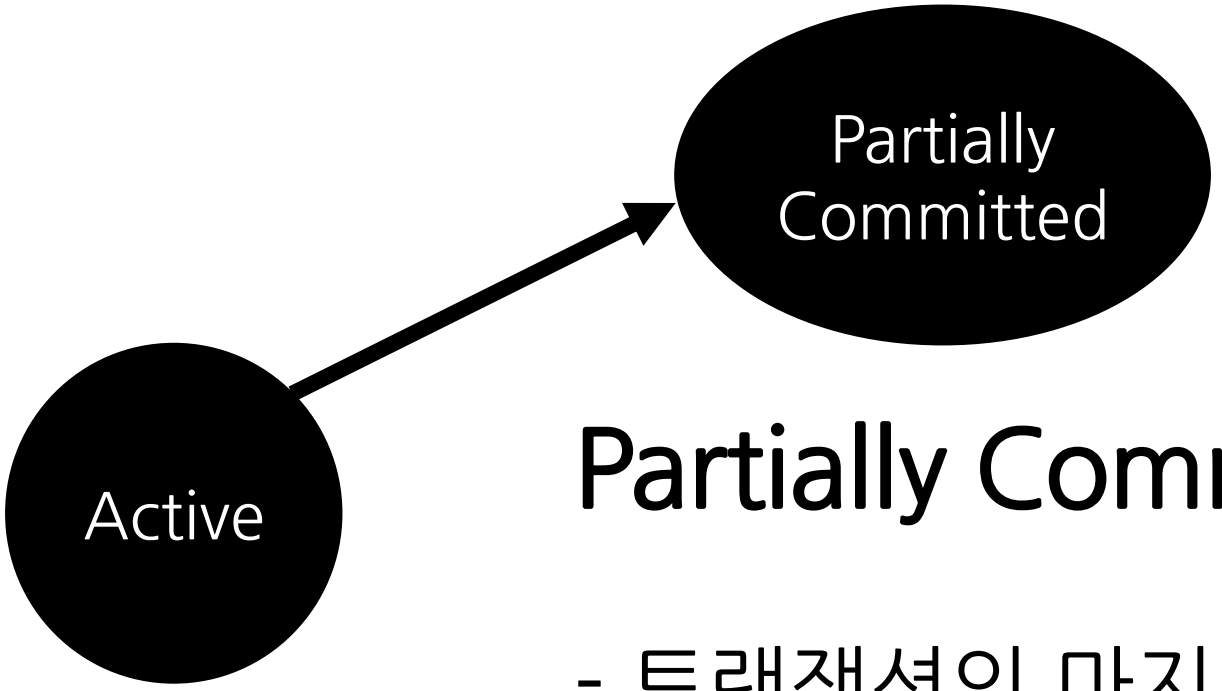
# 트랜잭션(작업)의 상태


## Active

- 트랜잭션의 초기 상태

- 실행 중에도 이 상태에 있다.

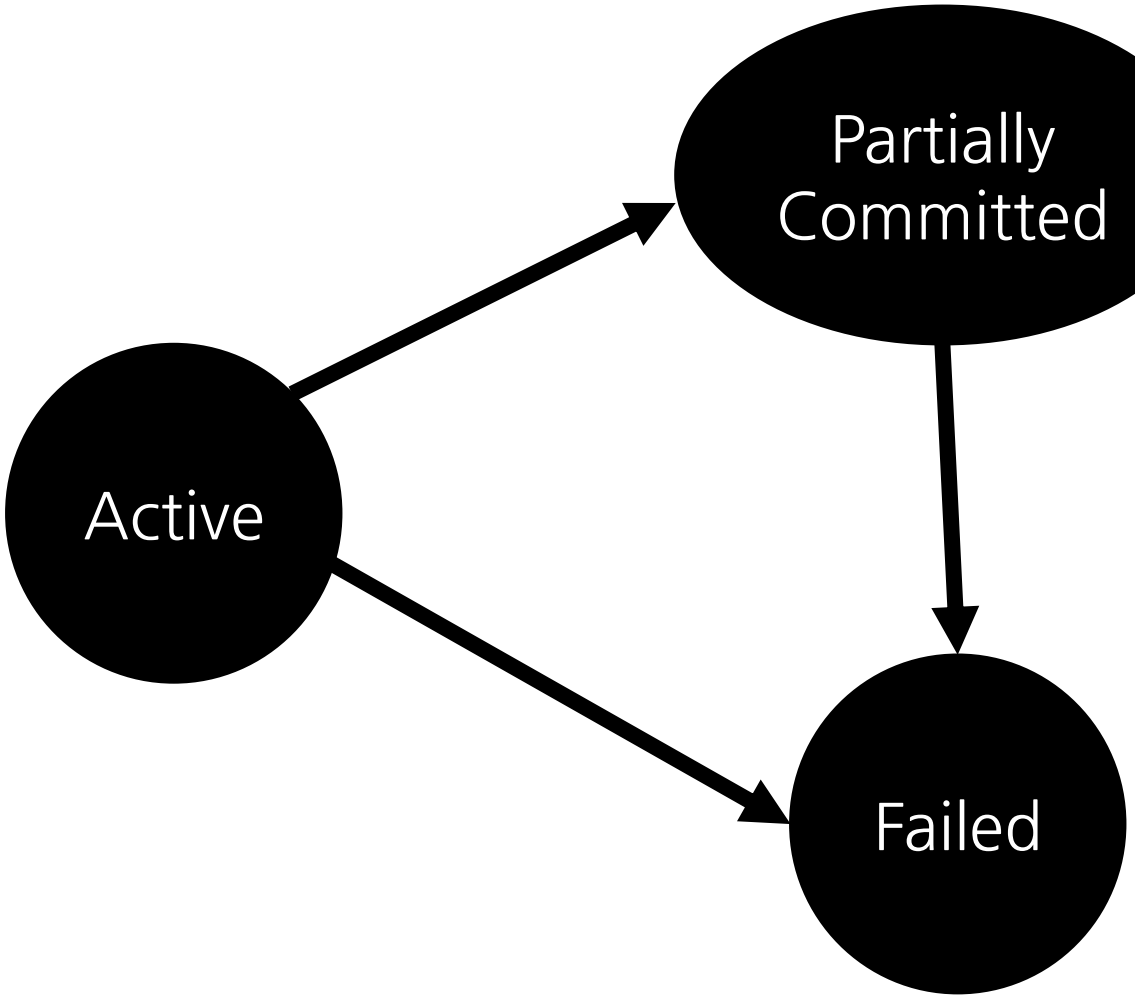
# 트랜잭션(작업)의 상태




## Partially Committed

- 트랜잭션의 마지막 연산이 끝난 직후
- 여기서 고장이 나면 Fail이 될 수 있다.

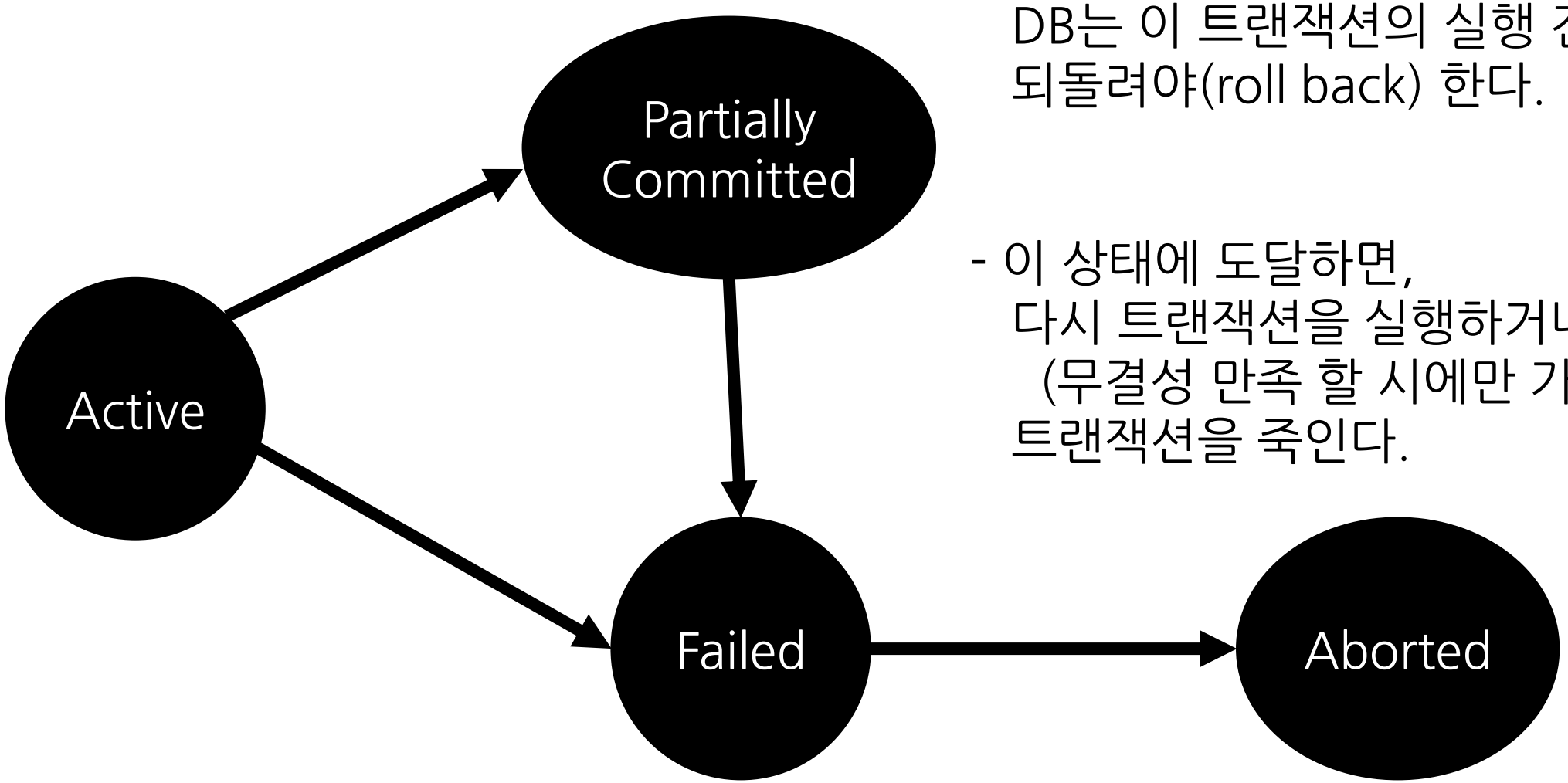
# 트랜잭션(작업)의 상태




## Failed

- 더 이상 정상적인 연산을 실행 할 수 없는 상태

# 트랜잭션(작업)의 상태

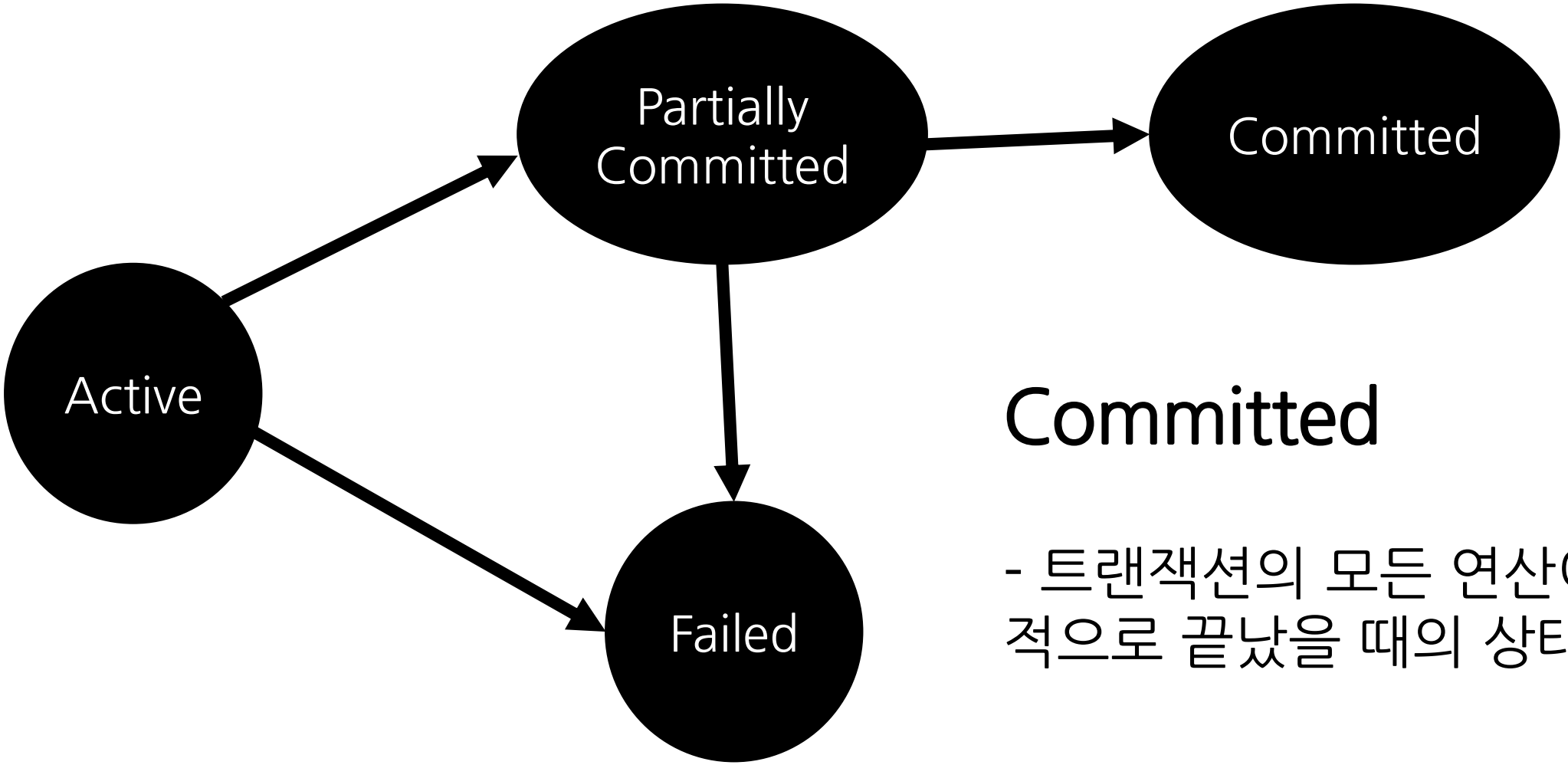



## Aborted

- 트랜잭션이 이 상태가 되면,
  DB는 이 트랜잭션의 실행 전 상태로
  되돌려야(roll back) 한다.

- 이 상태에 도달하면,
  다시 트랜잭션을 실행하거나
  (무결성 만족 할 시에만 가능)
  트랜잭션을 죽인다.

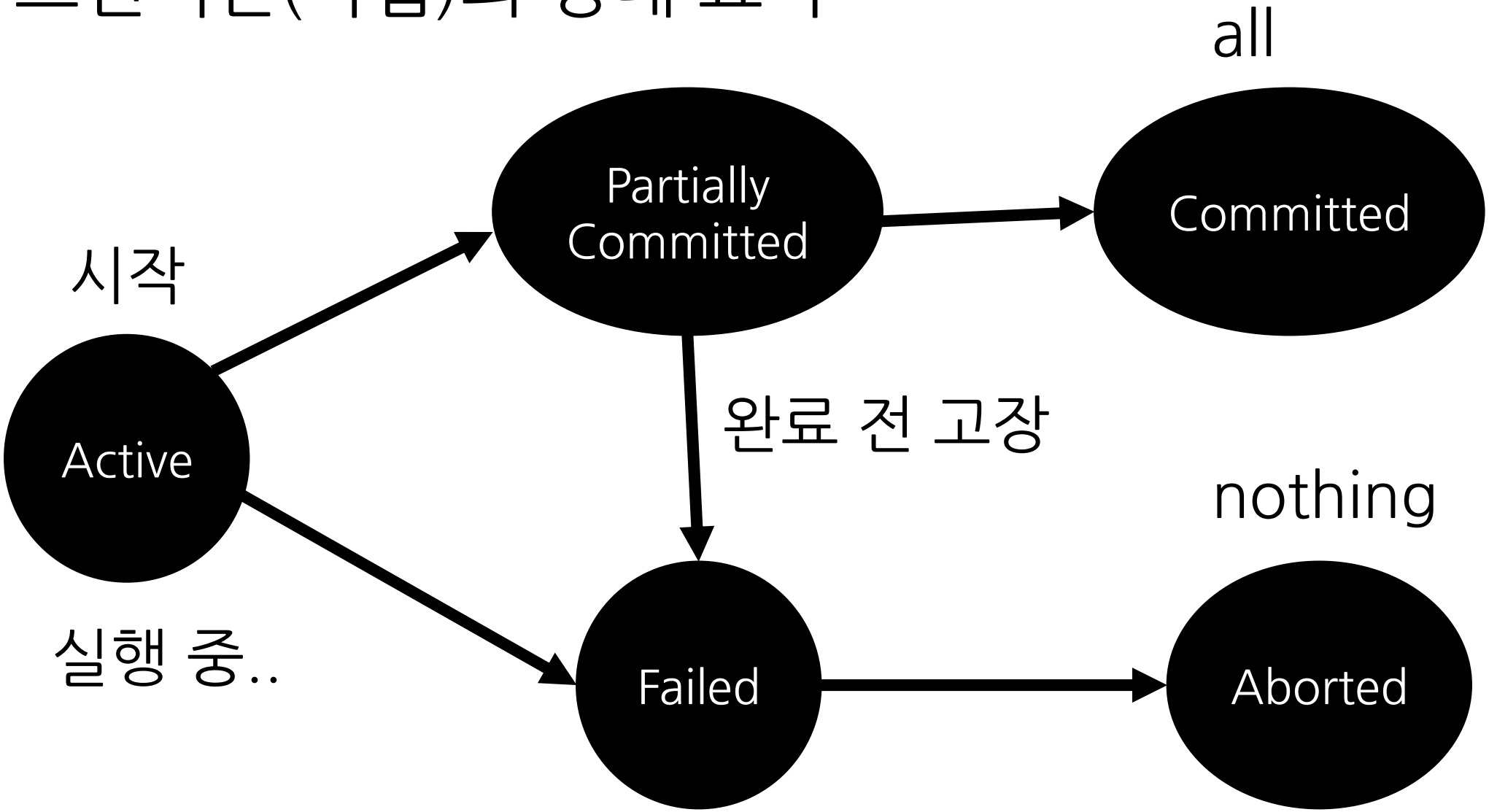
# 트랜잭션(작업)의 상태




## Committed

- 트랜잭션의 모든 연산이 성공적으로 끝났을 때의 상태

# 트랜잭션(작업)의 상태 요약



# 트랜잭션의 동시성(제일 어려움 집중!!!)

트랜잭션을 **순차적(Serial)**으로 실행하면 동시성 문제가 없지만,
동시에 트랜잭션을 실행하면 여러 이점이 있다.

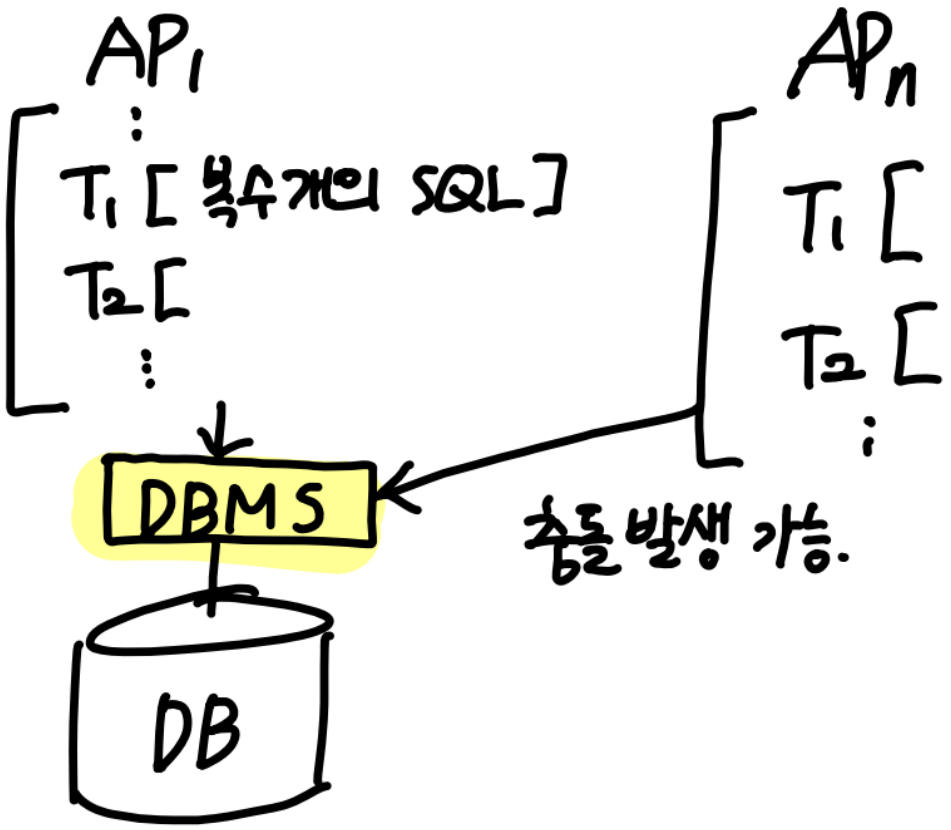
단위 시간 당 트랜잭션 처리량을 높일 수 있다.

응답 시간을 줄일 수 있다.

결국 정확성 VS 효율성의 싸움이다.

# 그럼 동시에 실행하면 되는 것 아닌가?

당연히, 문제(이상현상)가 생길 수 있다.



# 뭐가 문젠데?

운영체제에서의 문제와 똑같다.

1. 쓰는 동안 읽는 문제
2. 존재하지 않는 것을 접근하려는 문제
3. 동시에 쓰는 문제
4. 기타
5. + 연쇄적으로 롤백 등이 있는데 이런 것이 있구나! 하고 넘어감.

## 즉 DB의 일관성에 문제가 생길 수 있다.
(ACID의 C)

# 아니 근데….


그렇다고 동시성을 포기하기에는 **너무 느리다**.

동시성에서 발생 할 수 있는 여러 상황을 분류하고,
**동시성을 허용 할 수 있는 경우를 우리가 선택**하면 되지 않을까?

즉,
동시에 여러 트랜잭션을 실행하여 효율성을 높이면서,
순차적(Serializable)으로 실행 했을 때와 비슷한 결과를 낼 수 없을까?

# 이를 알아보기 전에..

책이나 교재, 인터넷에 있는 내용들은 전부
SQL92에서 발표한 isolation level 표준을 정의한 내용이다.

그런데, 마이크로소프트 연구원들이 낸
[A Critique of ANSI SQL isolation Levels]란 논문에는
정의한 표준이 모호하고, 실무에서 발생하는 이상현상을 다루지 않는다고 비판한다.

따라서, 설명을 할 때 표준을 먼저 설명하고, 부가 설명을 또 하는 식으로 발표를 할 것이므로

**설명이 모호하고 억지다 -> 그만큼 표준 예시가 개떡같다. (표준을 잘 만들어야 하는 반례)**
**이거 책에 없는 내용인데? -> 실무에서 발생하는 문제다. 몰라도 ㄱㅊ but 알아 두면 좋다**

라고 생각하면 편하다.

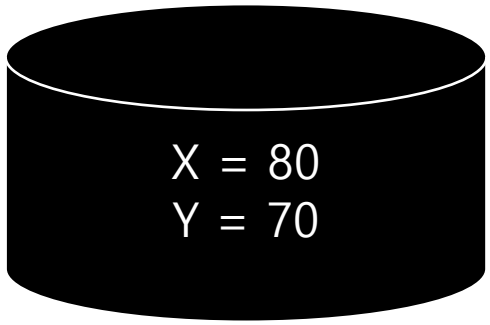
# 동시성을 허용 할 시 발생 할 수 있는 이상현상

1. Dirty Read (표준) :

Commit 되지 않은 데이터를 읽을 때 발생 할 수 있는 이상 현상

T1 :

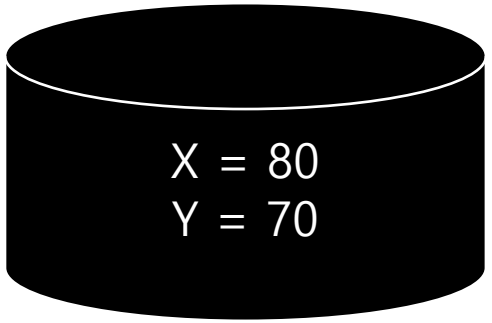
X에 y를 더한다.




T2 :

Y를 60으로 바꾼다.

# 이상현상1 : Dirty Read




**실행 흐름**

T1 :
X에 y를 더한다.

T2 :
Y를 60으로 바꾼다.

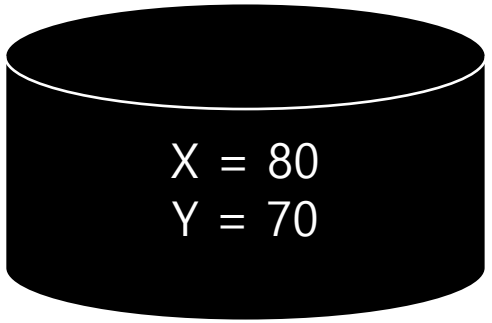
Read(X) : 결과 80

Write(y=60)

Read(Y) :결과 60
Write(X=140)
commit

Abort

# 이상현상1 : Dirty Read




**실행 흐름**

T1 :
X에 y를 더한다.

T2 :
Y를 60으로 바꾼다.

Read(X) : 결과 80

Write(y=60) 2. 그럼 얘도 잘못 된 값이게 된다.

Read(Y) :결과 60
Write(X=140) 3. 그럼 잘못 된 값을 더해서 나온 X도 잘못 된 값이다.
commit

Abort : 1. rollback y=70

# 동시성을 허용 할 시 발생 할 수 있는 이상현상

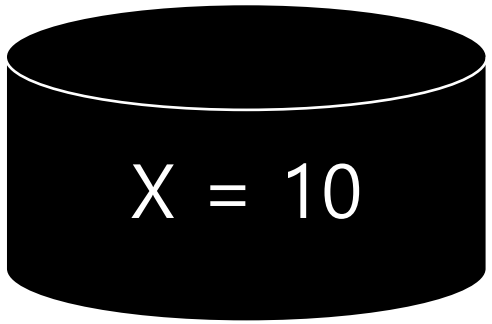
2. Non-repeatable Read (표준) :

같은 데이터의 값이 달라지는 이상현상

T1 :

X를 읽는다.

X를 읽는다.




T2 :

X에 40을 더한다.

# Non-repeatable Read

**실행 흐름**

T1 :
X를 읽는다 X를 읽는다

T2 :
X에 40을 더한다.


Read(X) : 10

Read(X) : 10
Write(X) : 50
commit

Read(X) : 50
commit

같은 데이터의 값이 달라짐

# 동시성을 허용 할 시 발생 할 수 있는 이상현상

## 3. Phantom Read (표준)

없는 데이터가 생기는 이상현상

T1 :

Product테이블에서 A조건을 만족하는 레코드들을 조회한다.

Product테이블에서 A조건을 만족하는 레코드들을 한번 더 조회한다.

T2 :

Product테이블에서 A조건을 만족하는 레코드를 삽입한다.

# Phantom Read

**실행 흐름**

T1 :

Product테이블에서 A조건을 만족하는 레코드들을 조회한다.

Product테이블에서 A조건을 만족하는 레코드들을 한번 더 조회한다. commit

T2 :

Product테이블에서 A조건을 만족하는 레코드를 삽입한다. commit


처음 조회에는 없던 레코드 발견!!!

3가지의 이상현상…

Dirty read

Non-repeatable read

Phantom Read

이 이상현상을 어느 정도까지 허용 할 것인가?

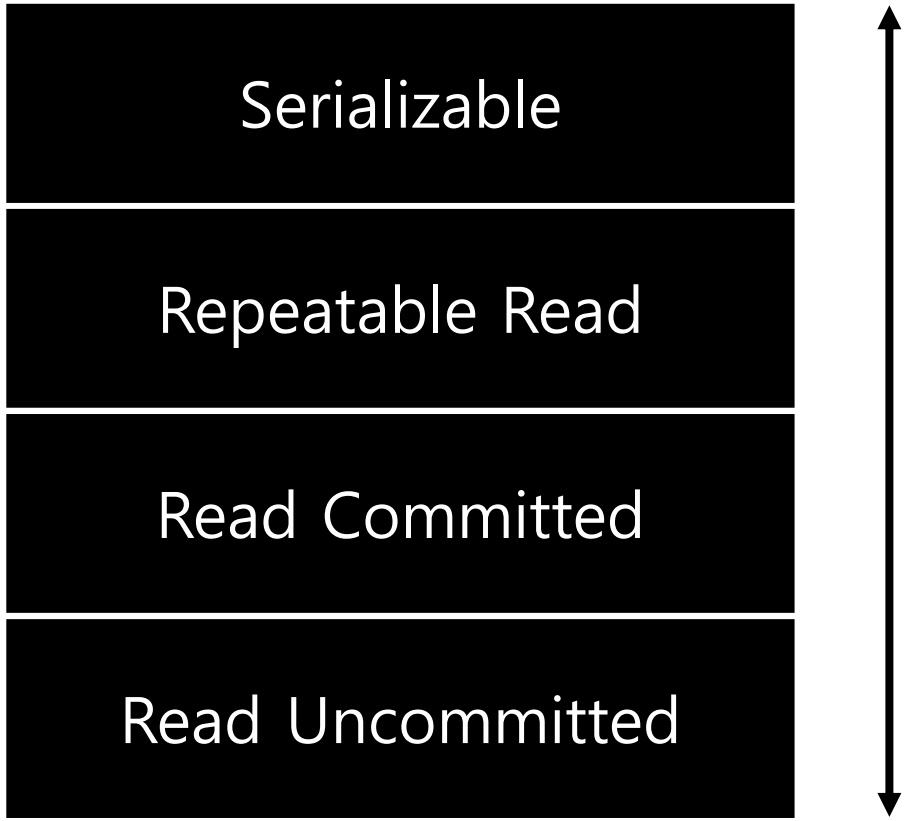
3가지의 이상현상…

“이걸 분류해서 사용자가 선택하게 하자”라
해서 나온 것이

Isolation Level

# Isolation level

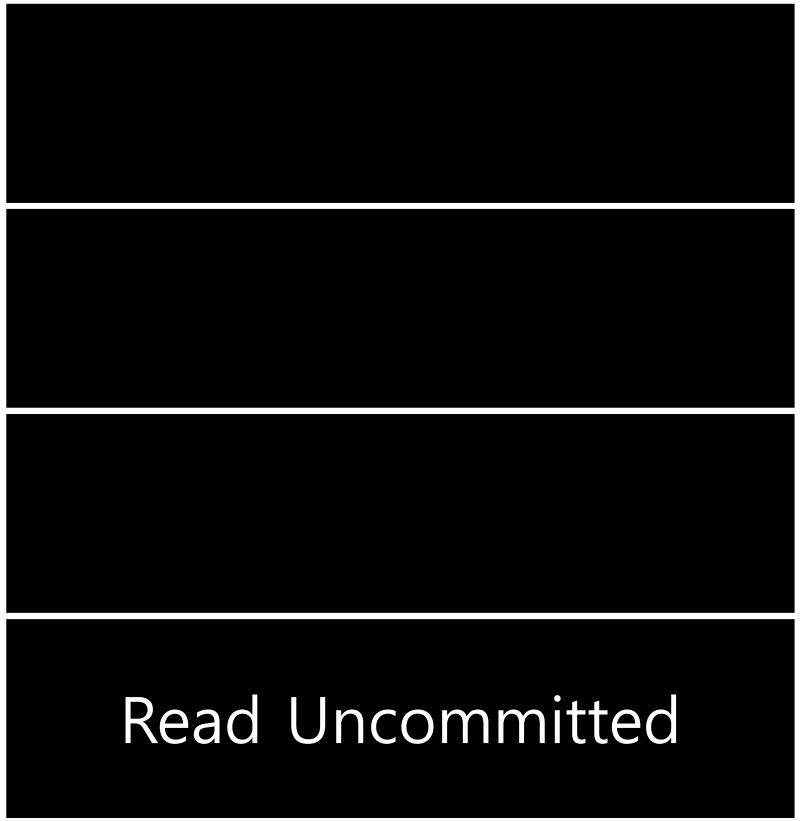
격리성, 일관성 높음, 효율성 낮음




격리성, 일관성 낮음, 효율성 높음

언 커 리 시

# Isolation level

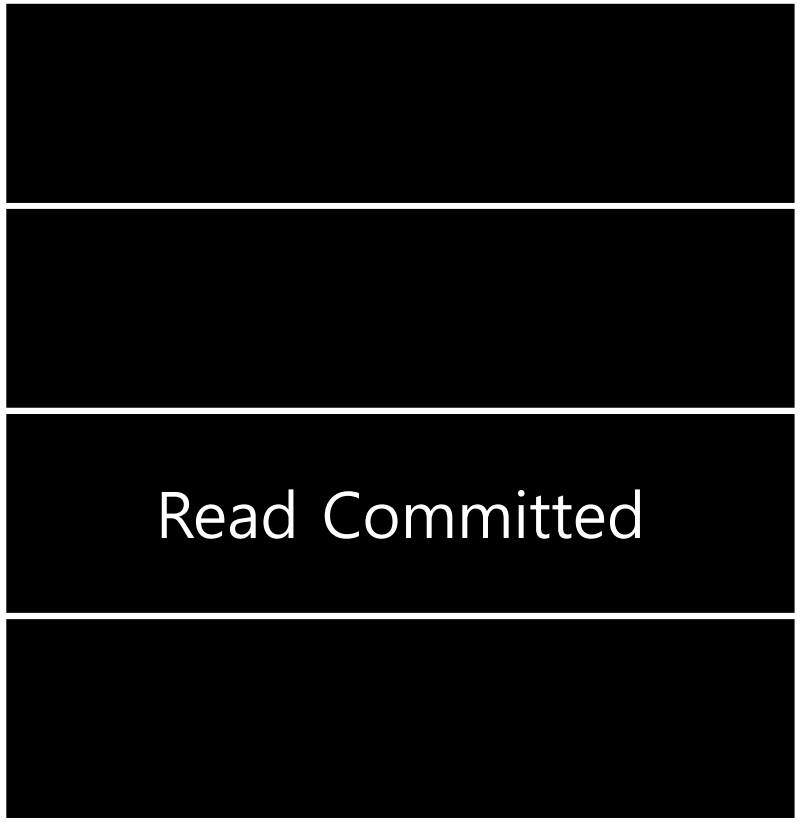



언 Read Uncommitted

제약 없는 모든 동시성 가능

| Dirty Read | Non-repeatable Read | Phantom Read |
|---|---|---|
| O | O | O |

# Isolation level



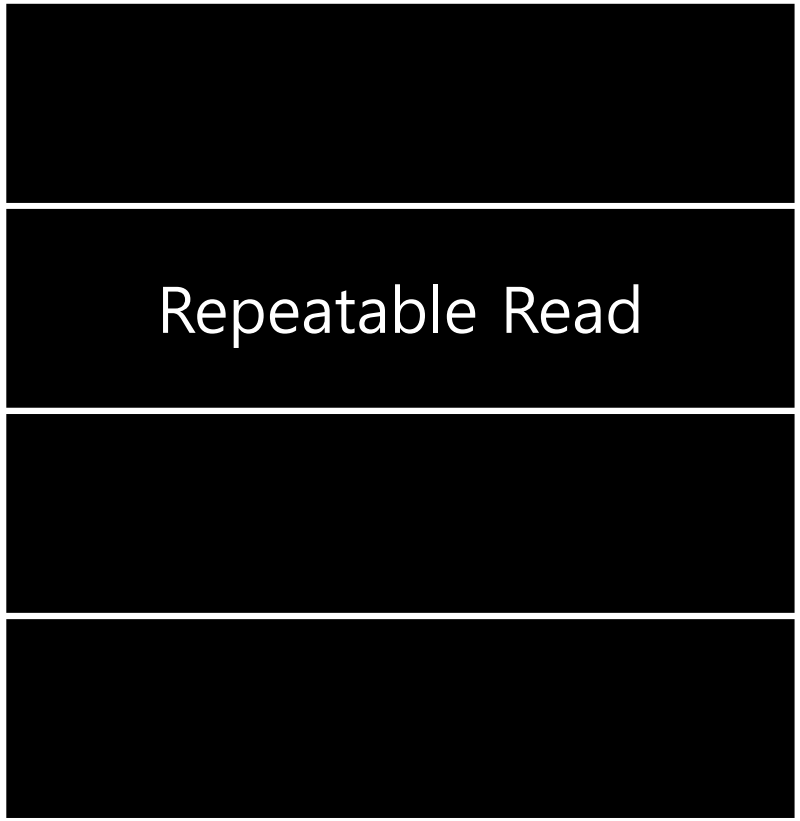

## 커 Read Committed

| Dirty Read | Non-repeatable Read | Phantom Read |
|---|---|---|
| X | O | O |

Read할 때 commit된 데이터만 가능.
But write에는 제약이 없어
Non-repeatable Read,
phantom read 발생 가능

# Isolation level

왜 그런지는 락을 알아야 하는데
락을 이해하려면 발만 살짝 담그는 것이 아닌
잠수를 해야 하므로 그냥 그렇구나~ 하고 이해

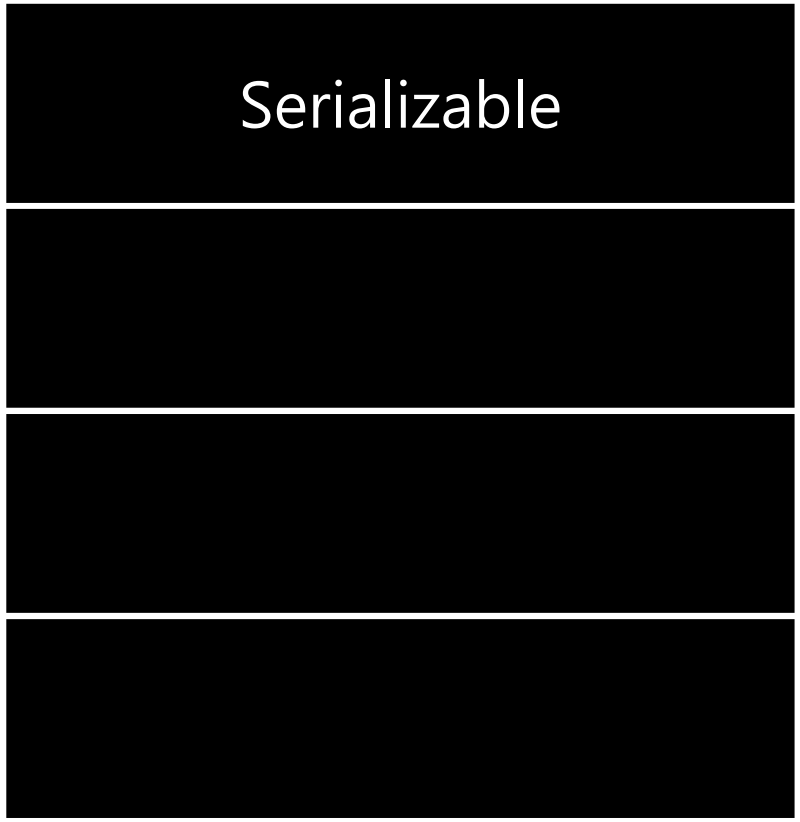



| Dirty Read | Non-repeatable Read | Phantom Read |
|---|---|---|
| X | X | O |

## 리 Repeatable Read

반복되는 read는 무조건 같은 결과를 내야 한다.(수정 한 데이터를 다른 트랜잭션이 수정하지 못하게 막음)
하지만 새로운 행을 추가하는 것은 상관하지 않아 Phantom read발생

# Isolation level




| Dirty Read | Non-repeatable Read | Phantom Read |
|---|---|---|
| X | X | X |

**시** Serializable

- 한 트랜잭션이 끝나야
  다음 트랜잭션이 실행

- 동시성 문제는 없지만 느리다.

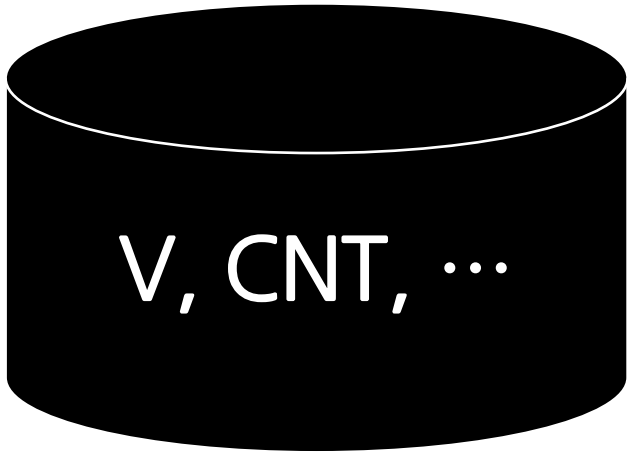
- 제일 정확한(일관적인) 상태

# But in real world..

1. 앞서 말한 세 가지 이상 현상의 정의가 모호
2. 또 다른 이상 현상들이 실무에서 발생
3. 표준이 실제 DBMS에서의 처리방법과 너무 달라.

# Phantom Read(추가 설명)

## 1. Phantom Read : 또 다른 상황(추가 설명)




어떤 데이터베이스에 이런 데이터가 있다고 가정하자.

V : 그냥 값
CNT : V의 값이 10을 넘는 레코드의 개수

# Phantom Read(추가 설명)

## 1. Phantom Read : 또 다른 상황(추가 설명)

T1 :

V>10인 데이터와
CNT를 읽는다.

T2 :

V=15인 데이터를 추가하고
CNT를 +1한다.

# Phantom Read(추가 설명)

**실행 흐름**

T1 :
V>10인 데이터와 CNT를 읽는다.

T2 :
V=15인 데이터를 추가하고 CNT를 +1한다.

Read(v>10): 결과 없음

Write(insert v=15)
Read(CNT) : 결과 0
Write(CNT = 1)
commit

Read(CNT) : 결과 1

commit

데이터 불일치 발생

같은 데이터가 아니더라도

연관된 데이터에서도

Phantom read 가 발생

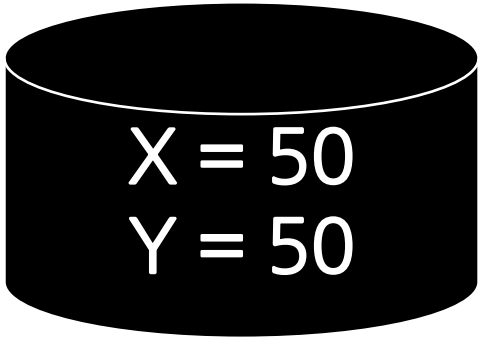
# 또 다른 이상 현상

## 2. Dirty Read(추가 설명) :

Commit되지 않은 데이터를 읽을 때 일어날 수 있는 이상 현상,
그러나 Rollback이 일어나지 않아도 발생 할 수 있다.

T1 :

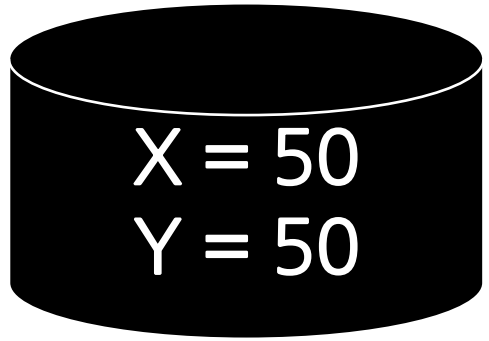
X가 Y에 40을 이체한다.




T2 :

X와 y를 읽는다.

# Dirty Read(추가 설명)




**실행 흐름**

T1 :
X가 Y에 40을 이체한다.

Read(x) : 50
Write(x = 10)

T2 :
X와 y를 읽는다.

Read(x) : 10
Read(y) : 50
commit

X+y = 60 ?!!!
데이터 불일치

Read(y) : 50
Write(y) : 90
commit

원래 x+y = 100
커밋 후 x+y = 100
정합성 ㅇㅋ

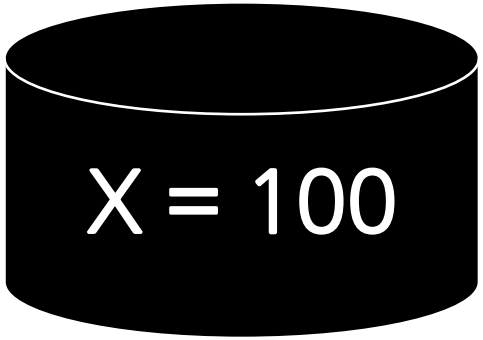
# 또 다른 이상 현상

## 3. Dirty Write :

commit안 된 데이터를 write 할 때의 이상 현상

T1 :

X를 10으로 바꾼다.




T2 :

X를 100으로 바꾼다.

# Dirty Write

**실행 흐름**

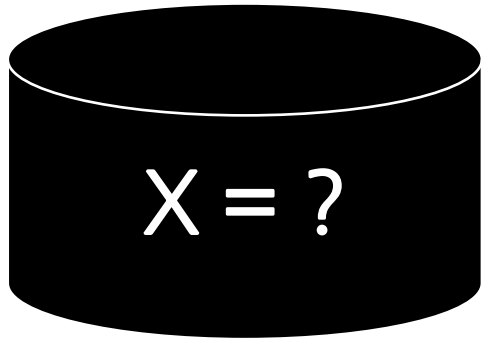
T1 :
X를 10으로 바꾼다.

Write(x = 10)

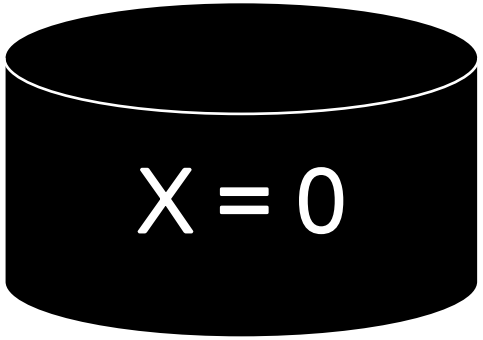
abort

T2 :
X를 100으로 바꾼다.

Write(x = 100)
commit



# Dirty Write




**실행 흐름**

T1 :
X를 10으로 바꾼다.

Write(x = 10)

T2 :
X를 100으로 바꾼다.

Write(x = 100)
commit

Isolation만큼 중요 한 것이 recovery
이므로
이런 이상현상은
그 어떤 level에서도 허용하면 안된다.

2. 이미 커밋이 된 트랜잭션임에도
불구하고 DB의 X는 다시 0이 됨.

Abort : 1. 이 트랜잭션에서는
바꾸기 전의 값이 0이므로 0으로 롤백

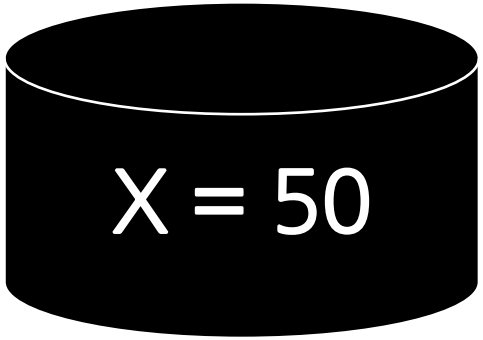
# 또 다른 이상 현상

## 4. Lost Update :

Update를 덮어 쓰는 이상 현상

T1 :

X에 50을 더한다.




T2 :

X에 150을 더한다.

# Lost Update

**실행 흐름**

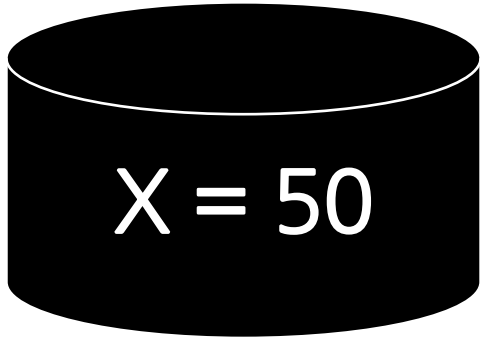
T1 :
X에 50을 더한다.

Read(x) : 50

Write(x = 100)
Commit
결과적으로 T2의 업데이트가
반영이 안 되었다.

T2 :
X에 150을 더한다.

Read(x) : 50
Write(x = 200)
commit




현욱이와 병선이가 나에게 돈을 보낼 때,

현욱이의 업데이트가 병선이의 업데이트를 지워버려서

병선이는 돈을 보냈는데 돈이 사라짐;;

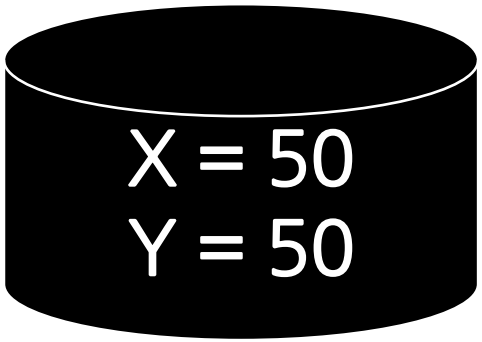
# 또 다른 이상 현상

## 5. Read Skew :

Inconsistent한 데이터 읽기

T1 :

X가 Y에 40을 이체한다.




T2 :

X와 y를 읽는다.

# Read Skew

**실행 흐름**

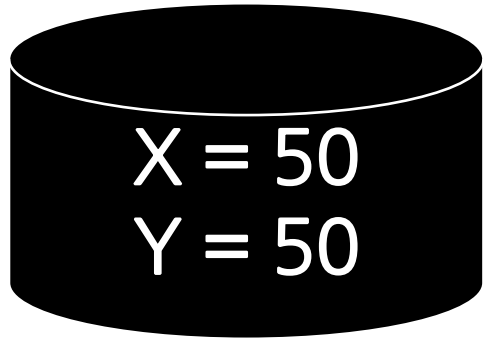
T1 :
X가 Y에 40을 이체한다.

Read(x) : 50
Write(x = 10)
Read(y) : 50
Write(y) : 90
commit

T2 :
X와 y를 읽는다.

Read(x) : 50

Read(y) : 90
commit




X + y = 140 ?!

Dirty read와 다르게,
Commit된 데이터를
읽었음에도

데이터 불일치 발생

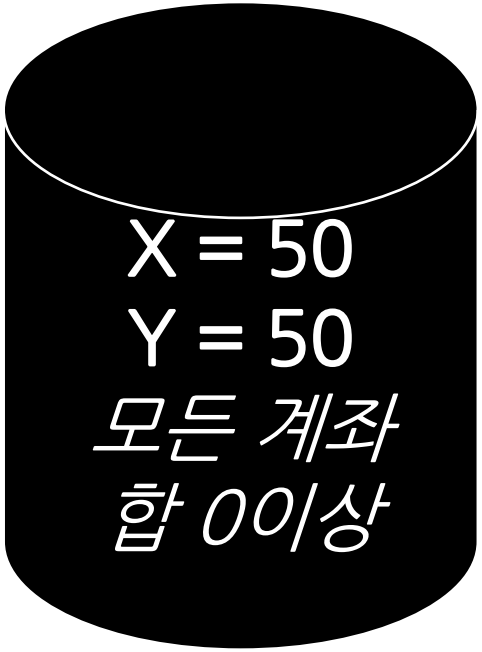
# 또 다른 이상 현상

## 6. Write Skew :

Inconsistent한 데이터 읽기

T1 :

X에서 80 인출




T2 :

Y에서 90인출

# Write Skew

**실행 흐름**

T1 :
X에서 80 인출.

Read(x) : 50
Read(y) : 50

Write(x = -30)

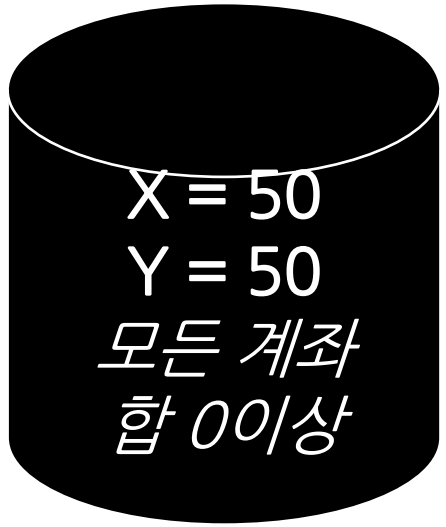
commit

T2 :
Y에서 90인출

Read(x) : 50
Read(y) : 50

Write(y= -40)

commit




X+y >=0 을 만족하는지 확인하기 위해 x,y둘 다 읽기

그 다음 버퍼에서 각 트랜잭션에서 조건을 만족하므로 인출 후 커밋

# Write Skew

**실행 흐름**

T1 :
X에서 80 인출.

Read(x) : 50
Read(y) : 50

Write(x = -30)

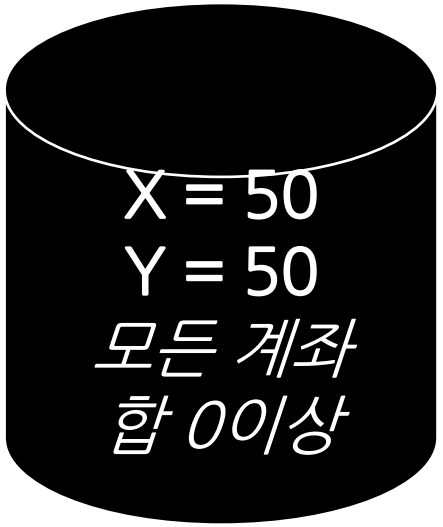
commit

T2 :
Y에서 90인출

Read(x) : 50
Read(y) : 50

Write(y= -40)

commit




두 트랜잭션 실행 후
X = -30
Y = -40

즉 x+y >=0의 조건에 위배

서로 다른 데이터를 썼음에도
데이터 불일치 문제 발생

# 요약

Dirty Read, Non-repeatable read, phantom read의 이상 현상만 있는 줄 알았는데,

실제로는 다른 상황에서 표준의 상황이 일어 날 수 있었으며,
dirty write, lost update, read skew, write skew 등 여러 이상현상이 있었다!!!!

# 요약

그럼 실제 DBMS에서는 어떤 isolation level을 사용?

Snapshot isolation

# Snapshot isolation?

원래 isolation level표준은

이상 현상을 분류하고, 어느 정도로 허용 할 것인지를 level로 구분

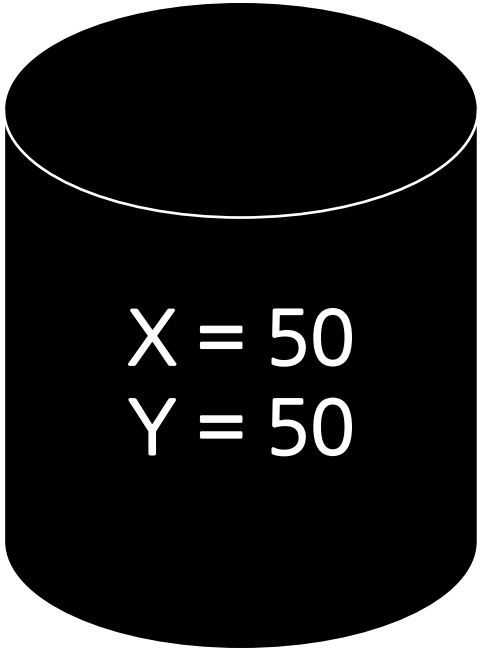
그런데 snapshot isolation은 동시성 컨트롤의 구현에 따라 level을 정의

실제로 어떤 식으로 동작 할 까?

# 발을 살짝 담가보자

T1 :

X가 y에 40인출



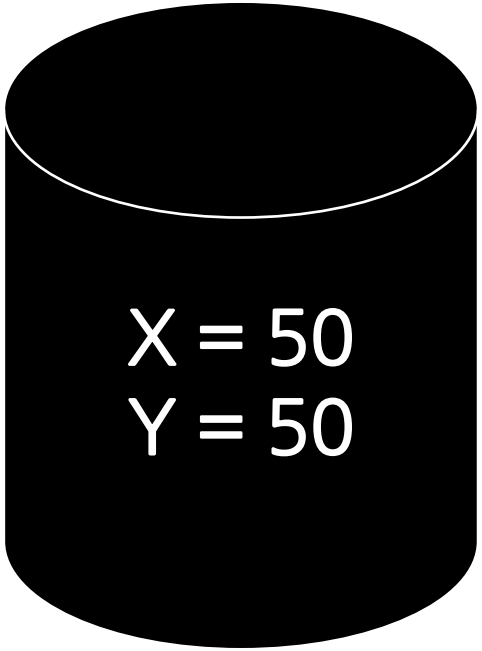

T2 :

Y에 100 입금

# 발을 살짝 담가보자

T1 :

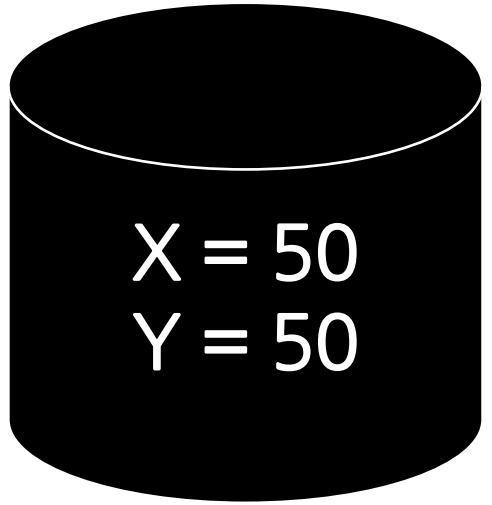
X가 y에 40인출




T2 :

Y에 100 입금

# Snapshot Isolation




**실행 흐름**

T1 :
X가 y에 40인출.

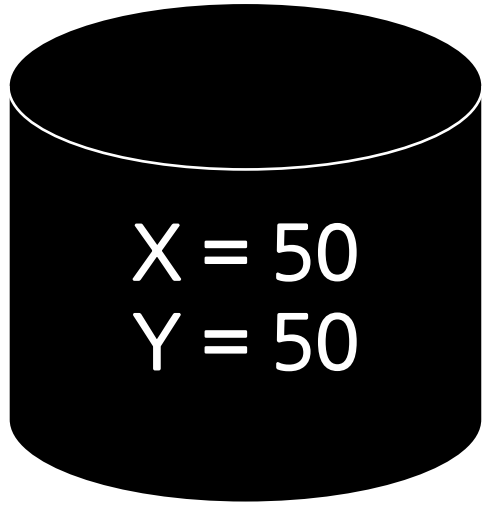
Read(x) : 50
write(x = 10)

T2 :

트랜잭션이 시작 할 때
db의 스냅샷을 찍는다

T1 snapshot

# Snapshot Isolation




**실행 흐름**

T1 :
X가 y에 40인출.

T2 :
Y에 100 입금

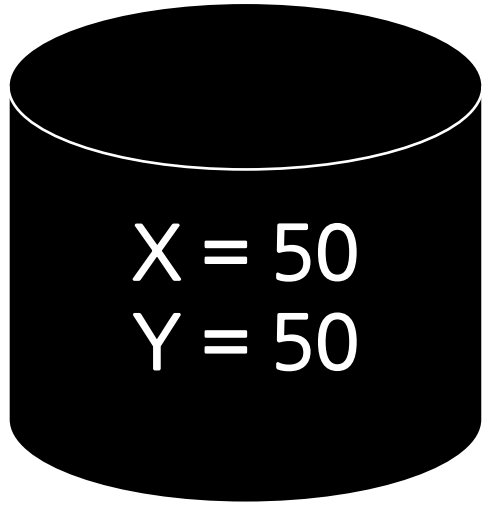
Read(x) : 50
write(x = 10)

이때 write를 db에 하는 것이 아닌,
스냅샷에 저장!

T1 snapshot

X = 10

# Snapshot Isolation




**실행 흐름**

T1 :
X가 y에 40인출.

Read(x) : 50
write(x = 10)

T2 :
Y에 100 입금

T2 snapshot

Y = 150

Read(y) : 50
Write(y = 150)
commit

이때 write를 db에 하는 것이 아닌,
T2의 스냅샷이라는 공간에 저장!

T1 snapshot

X = 10

# Snapshot Isolation

**실행 흐름**

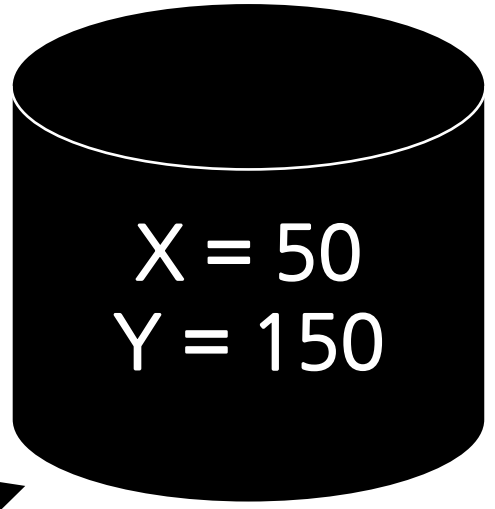
T1 :
X가 y에 40인출.

Read(x) : 50
write(x = 10)

T2 :
Y에 100 입금

T2 snapshot

Y = 150



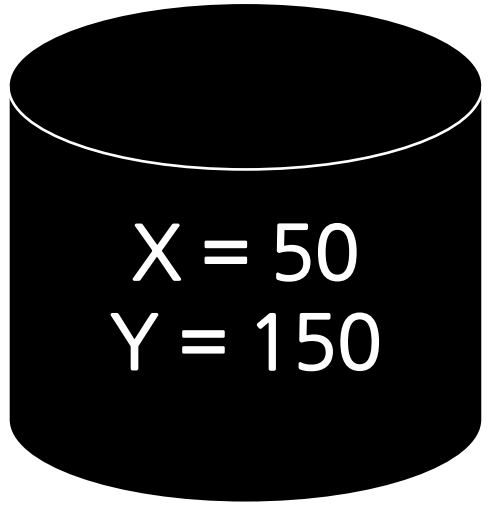

Read(y) : 50
Write(y = 150)
commit

커밋을 할 때
스냅샷에서 DB로 저장!

T1 snapshot

X = 10

# Snapshot Isolation




**실행 흐름**

T1 :
X가 y에 40인출.

T2 :
Y에 100 입금

Read(y) : 50
Write(y = 150)

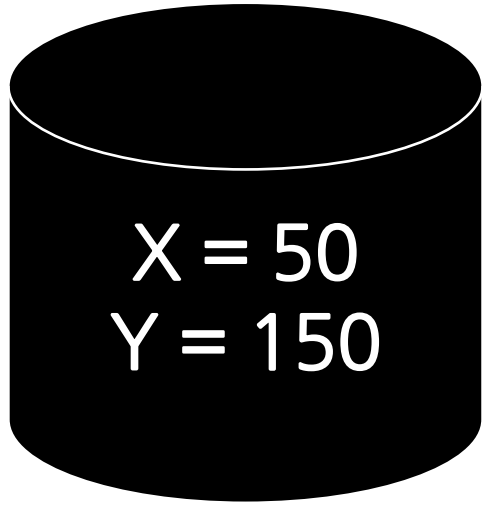
이제 y에 돈을 더함
Read(y) : 50
Write(y = 90)

Db에 150이 저장되었다고 150을 읽을까? ㄴㄴ
T1의 스냅샷을 찍은 시점의 y인 50을 읽는다!!!!

T1 snapshot

X = 10

# Snapshot Isolation




**실행 흐름**

T1 :
X가 y에 40인출.

T2 :
Y에 100 입금

Read(y) : 50
Write(y = 150)
commit

이제 y에 돈을 더함
Read(y) : 50
Write(y = 90)

마찬가지로 db에 write하는 것이 아닌,
T1의 스냅샷에 저장!

T1 snapshot

X = 10, y = 90

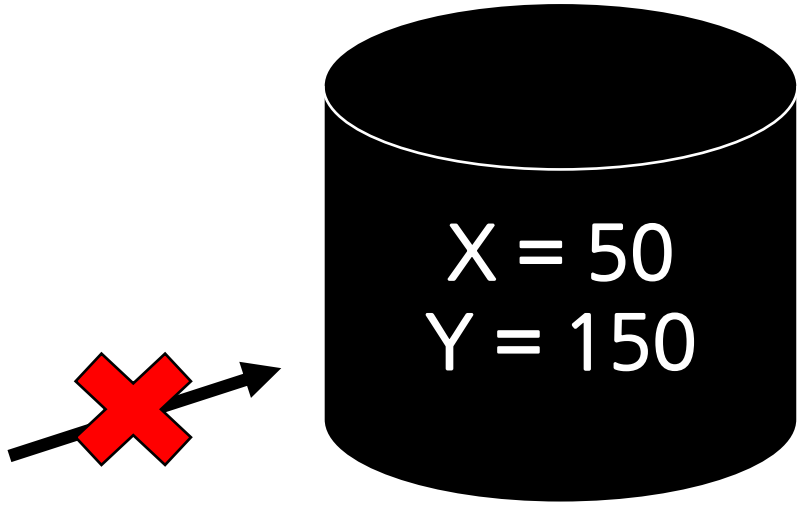
# Snapshot Isolation

**실행 흐름**

T1 :
X가 y에 40인출.

T2 :
Y에 100 입금

Read(y) : 50
Write(y = 150)
commit



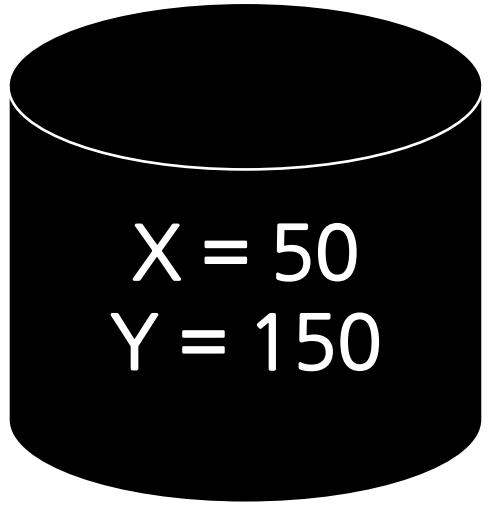

T1 snapshot

X = 10, y = 90

이제 y에 돈을 더함
Read(y) : 50
Write(y = 90)
commit

이때 스냅샷을 바로 db에 저장하면 T2의 결과가 사라지는
Lost update가 일어 날 것이다!

# Snapshot Isolation




**실행 흐름**

T1 :
X가 y에 40인출.

이제 y에 돈을 더함
Read(y) : 50
Write(y = 90)
Commit -> abort

T2 :
Y에 100 입금

Read(y) : 50
Write(y = 15
commit

Snapshot isolatio에서는 같은 데이터의 write가 일어날 때,
먼저 커밋 된 변경만 인정 해 준다!

그러면 나중에 commit한 트랜잭션은 abort 처리가 된다!!!
그럼 Abort일 때는 어떻게 되나요?
해당 스냅샷을 삭제하면 끝난다.

# Snapshot Isolation

지금 설명한 스냅샷 아이솔레이션은

MVCC, Multi Version Concurrency Control의 한 종류다
각 트랜잭션마다 특정 시점의 스냅샷을 찍는다.

따라서 트랜잭션 시작 전 커밋된 데이터만 보이며,
같은 데이터에 대한 write시 먼저 커밋 한 트랜잭션이 이기는
First-Committer win이라는 전략을 따른다.

# DBMS에서의 isolation level

그럼 실제 DBMS에서는 이 레벨을 어떻게 따르고 있을까?

# DBMS : (Mysql)innoDB : Repeatable Read

MySQL 8.0 Reference Manual / ... / Transaction Isolation Levels

version 8.0

### 15.7.2.1 Transaction Isolation Levels

Transaction isolation is one of the foundations of database processing. Isolation is the I in the acronym ACID; the isolation level is the setting that fine-tunes the balance between performance and reliability, consistency, and reproducibility of results when multiple transactions are making changes and performing queries at the same time.

InnoDB offers all four transaction isolation levels described by the SQL:1992 standard: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE. The default isolation level for InnoDB is REPEATABLE READ.

언 커 리 시 표준을 그대로 따른다.

**동시성 제어**

다중 버전 동시성 제어(MVCC)는 레코드의 중복 사본을 생성하여 동일한 데이터를 병렬로 안전하게 읽고 업데이트하는 고급 데이터베이스 기능입니다. MVCC를 사용하면 여러 사용자가 데이터 무결성을 손상시키지 않고 동일한 데이터를 동시에 읽고 수정할 수 있습니다.

MySQL 데이터베이스는 MVCC를 제공하지 않지만 PostgreSQL은 이 기능을 지원합니다.

# DBMS : Oracle : Read Committed

## Oracle Isolation Levels

Oracle provides these transaction isolation levels.

| Isolation Level | Description |
|---|---|
| Read committed | This is the default transaction isolation level. Each query executed by a transaction sees only data that was committed before the query (not the transaction) began. An Oracle query never reads dirty (uncommitted) data.<br><br>Because Oracle does not prevent other transactions from modifying the data read by a query, that data can be changed by other transactions between two executions of the query. Thus, a transaction that runs a given query twice can experience both nonrepeatable read and phantoms. |
| Serializable | Serializable transactions see only those changes that were committed at the time the transaction began, plus those changes made by the transaction itself through `INSERT`, `UPDATE`, and `DELETE` statements. Serializable transactions do not experience nonrepeatable reads or phantoms. |
| Read-only | Read-only transactions see only those changes that were committed at the time the transaction began and do not allow `INSERT`, `UPDATE`, and `DELETE` statements. |

표준의 RC, S를 따르며, 다른 read only라는 격리 레벨도 사용

# DBMS : PostgreSQL : Read Committed

Table 13.1. Transaction Isolation Levels

| Isolation Level | Dirty Read | Nonrepeatable Read | Phantom Read | Serialization Anomaly |
|---|---|---|---|---|
| Read uncommitted | Allowed, but not in PG | Possible | Possible | Possible |
| Read committed | Not possible | Possible | Possible | Possible |
| Repeatable read | Not possible | Not possible | Allowed, but not in PG | Possible |
| Serializable | Not possible | Not possible | Not possible | Not possible |

In PostgreSQL, you can request any of the four standard transaction isolation levels, but internally only three distinct isolation levels are implemented, i.e., PostgreSQL's Read Uncommitted mode behaves like Read Committed. This is because it is the only sensible way to map the standard isolation levels to PostgreSQL's multiversion concurrency control architecture.

표준을 따르지만, Read uncommitted는 committed처럼 행동
그리고 마이크로소프트 논문에서 다룬 표준에 없는 이상 현상들은 Serialization Anomaly라는 항목으로 분류 및 레벨 설정

# 그럼 nosql은?

MongoDB

https://www.mongodb.com › docs › manual › core › r...

## Read Isolation, Consistency, and Recency

**Read uncommitted is the default isolation level** and applies to mongod standalone instances as well as to replica sets and sharded clusters. Read Uncommitted ...

# 그래서 어떤 dbms를 써?

내 프로젝트의 요구사항과 기본 격리성 레벨이 맞는 디비를 선택하자!

END

# 선택) 심화 : 동시성, 어떻게 구현하는가

이제부터는 쿼리가 아닌, DBMS의 입장에서 트랜잭션을 해결하는 방법을 본다.

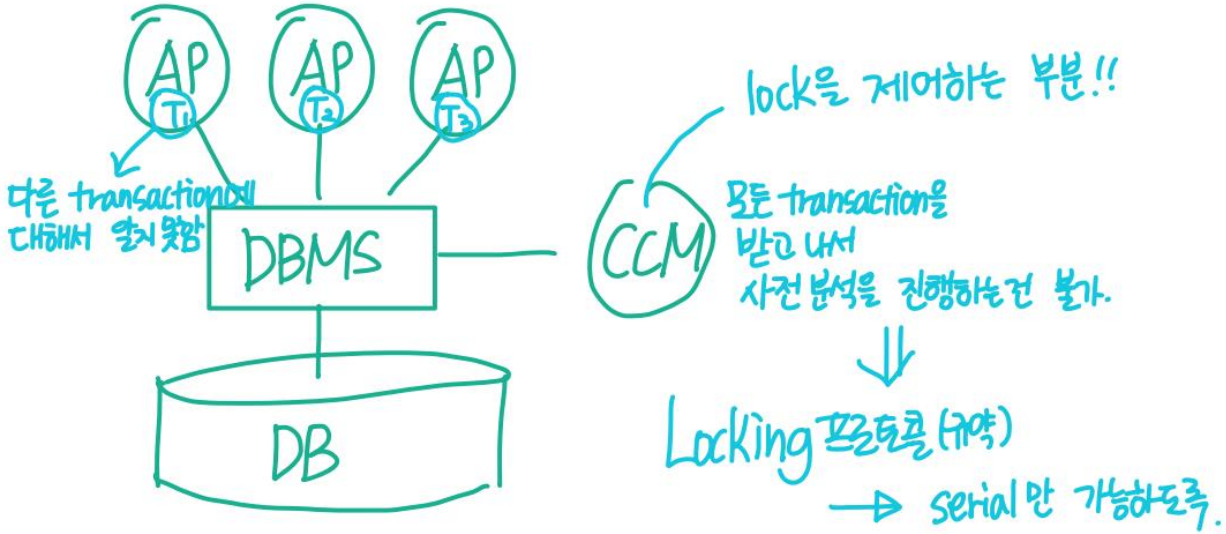
시간이 애매하면 패스, 막상 쓰다 보니 cs책에 없어서 선택으로 남김. ㅠㅠ

# 동시성 제어를 구현하는 방법

## Locking
- 전체 DB를 잠글까 아니면 일부 데이터를 잠글까?
- 잠금은 얼마나 지속해야 할까?
- 공유 락 vs 배타 락

## 타임스탬프(생략)
- 강해져서 돌아와라.

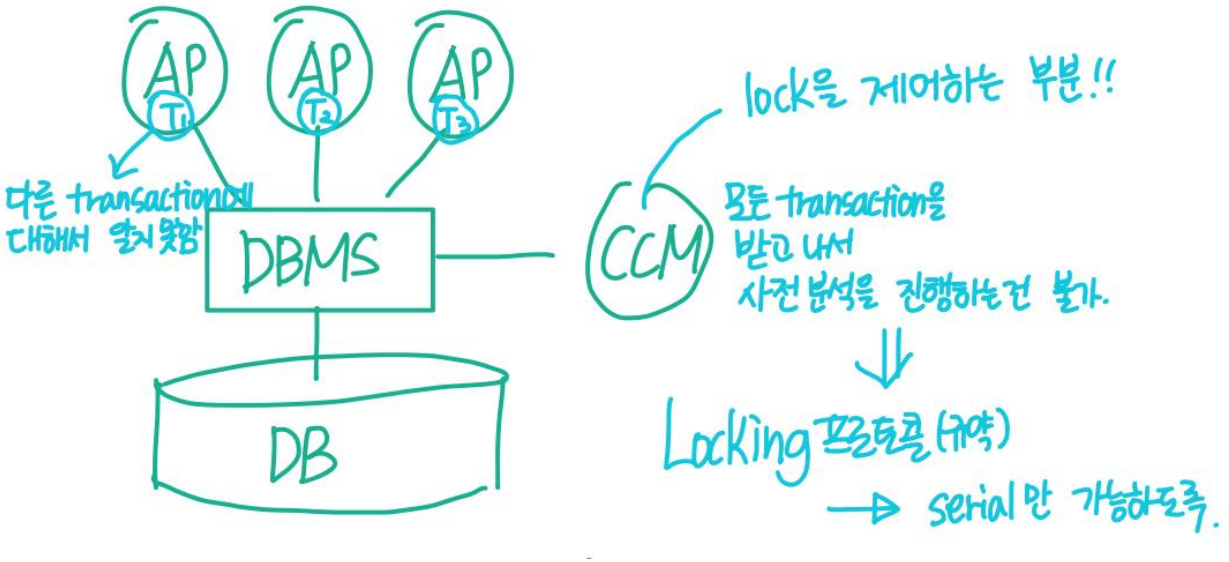
## 스냅샷(생략)
- 강해져서 돌아와라.

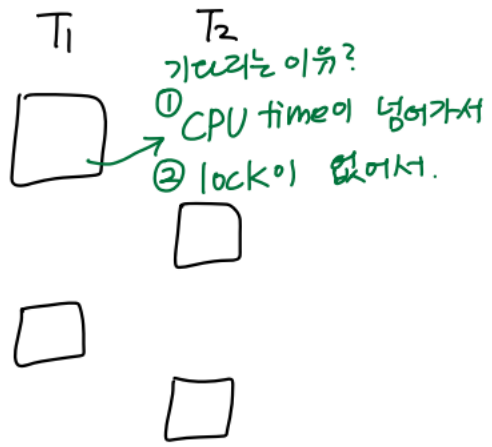



우리가 이걸 다 해? ㄴㄴ
Concurrency Control Manager가 해.

# Lock

데이터에 접근 전, 동시성 제어 관리자(CCM)에게 lock을 얻을 수 있다.
**락 요청을 CCM이 허용 해줘야 연산 실행 가능**




즉 락이 필요한데 락을 걸 수 없는 상황이면? 이미 걸린 락이 끝날 때 까지 기다려야 한다.

# Lock, 이 데이터는 이제 제 겁니다.

데이터를 2가지 방법의 잠금을 걸 수 있다.

1. **Exclusive (X) mode  쓰기**
   - 이 데이터에 대한 read, write는 나만 할 수 있다. ㄹㅇ 독점
   - **lock-X** 라고 표기

2. **Shared (X) mode  읽기**
   - 다른 트랜잭션이 이 데이터를 read만 할 수 있다.
   - 그런데 나도 read만 할 수 있다.  공유
   - **lock-S** 라고 표기

# Lock

그렇다면 한 데이터에 대해 다른 트랜잭션들이 Lock을 걸 수 있어?
답) 그럴 수도 있고 아닐 수도 있다.

| data | Lock-X | Lock-S |
|---|---|---|
| Lock-X | 불가능 | 불가능 |
| Lock-S | 불가능 | 가능 |

# Lock 쉽게 이해하기

| data | Lock-X | Lock-S |
|---|---|---|
| Lock-X | 불가능 | 불가능 |
| Lock-S | 불가능 | 가능 |

배타 락- 쓰기  공유 락 - 읽기라고 생각하면 이해가 쉽다.

**데이터를 쓰고 있는데 읽어도 되나요? ㄴㄴ 일관성 문제가 생길 수 있어.**
**데이터를 읽고 있는데 써도(write) 되나요? ㄴㄴ 일관성 문제가 생길 수 있어.**
**데이터를 쓰고 있는데 써도(write) 되나요? ㄴㄴ 죽고 싶어?**
**데이터를 읽고 있는데 읽어도 되나요? ㅇㅇ.**

락이 필요한데
락을 걸 수 없는 상황이면?

이미 걸린 락이 끝날 때
까지 **기다려**야 한다.

# 그렇다면 lock의 해제 시점은 언제 일까요?

1. 그 데이터의 read/write가 끝난 직후 일 것이다.
2. 이 전체 트랜잭션이 성공/실패 한 이후 일 것이다.

일단은 여기까지! 생각만 해보자.