Transaction

태그	DB
주차	2주차

목차

1. Transaction
    1-1. Transaction의 특징 (ACID)
        1) 원자성(Atomicity)
        2) 일관성(Consistency)
        3) 독립성(Isolation)
        4) 지속성(Durability)
    1-2. Transaction의 종류
    1-3. Transaction 연산 (TCL)
        1) Commit
        2) Rollback
    1-4. Undo/Redo
        1) Redo
        2) Undo
    1-5. Schedule(History)
        0) 스케줄 예시
        1) Conflict Serializable(충돌 직렬 가능 스케줄)
        2) View Serializable(뷰 직렬 가능 스케줄)
2. Isolation Level
    2-1. Isolation Level의 필요성
    2-2. Isolation Level의 종류
        0) Level 0: Read Uncommitted
        1) Level 1: Read Committed
        2) Level 2: Repeatable Read
        3) Level 3: Serializable
        +) Snapshot Isolation

1. Transaction

- **DB의 상태를 변화**시키기 위해서 수행하는 **작업의 단위**

📖 **DB의 상태변화**

아래 질의어(SQL)을 통해 DB에 접근하는 것

- SELECT
- INSERT
- DELETE
- UPDATE

📖 **작업 단위**

질의어 한 문장이 아님!

목적에 맞게 DB의 상태 변화를 유발하는 질의어들을 묶어 하나의 작업, 하나의 Transaction이라고 함

1-1. Transaction의 특징 (ACID)

1) 원자성(Atomicity)

- **All or Nothing**
  - 모두 반영되든가, 전혀 반영되지 않던가

📎 **이점**

- 시스템을 이해하기 쉬워짐
- 오작동시 원인이 되는 트랜잭션을 찾기 쉬워짐

2) 일관성(Consistency)

- **DB의 상태가 일관되어야 함**
  - 하나의 Transaction 이후에도, DB의 상태는 이전과 같이 유효해야 함
  - 즉, Transaction이 일어난 이후의 DB도 DB의 제약이나 규칙을 만족해야 함

📎 **이점**

- Transaction이 진행되는 중간에 DB가 변경되더라도, 변경 전 DB로 작업이 진행됨으로써 일관성있는 데이터를 볼 수 있음

3) 독립성(Isolation)

- Transaction은 상호 독립적으로 실행됨
  - 각 Transaction은 다른 Transaction의 존재를 알 수 없음
  - 각 Transaction은 마치 혼자만 실행되는 것 처럼 고립(격리)되어 실행됨
- 실제로는 여러 Transaction이 동시에 실행되더라도, 각각 연속으로 실행된 것과 같은 결과를 냄 (내야 함)

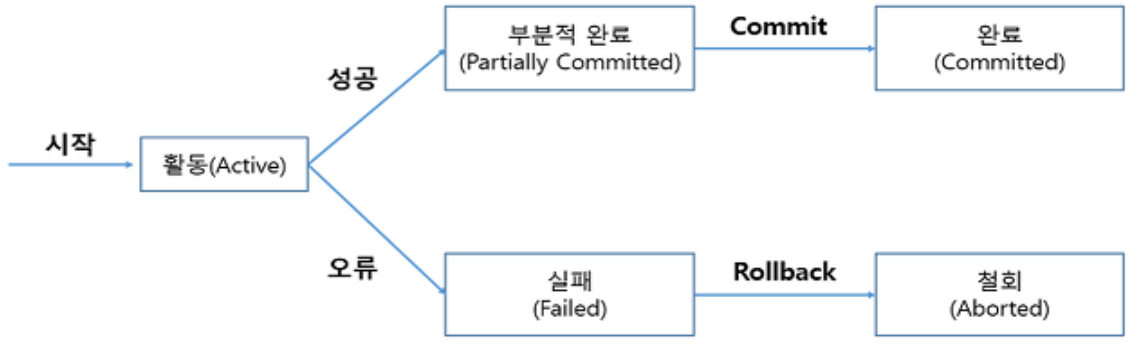
> 📎 **이점**
>
> - 사용자는 다른 사용자의 작업을 신경쓰지 않고 본인의 작업 가능

4) 지속성(Durability)

- Transaction이 성공적으로 완료됐을 경우, 그 결과(로그)가 영구적으로 반영돼야 함
  - 런타임 및 시스템 오류가 발생하더라도, 그 기록이 영구적이어야 함

> 📎 **이점**
>
> - DB를 신뢰도 있게 사용 가능

1-2. Transaction의 종류



트랜잭션의 상태

1. **활동(Active)**
   - Transaction이 정상 실행중인 상태

2. **부분적 완료(Partially Committed)**
    - Commit 연산 직전의 상태로, 결과가 저장되지 않은 상태
3. **완료(Committed)**
    - Transaction이 성공적으로 종료되어 Commit 연산을 실행한 후의 상태
4. **실패(Failed)**
    - Transaction이 오류로 인해 중단된 상태
5. **철회(Aborted)**
    - Transaction이 비정상 종료로 인해 Rollback 연산을 수행한 상태

1-3. Transaction 연산 (TCL)

1) Commit

- Transaction의 종료를 알리는 연산
- Transaction이 성공적으로 끝났고, DB가 일관성있는 상태일 때 실행
- 수행했던 Transaction이 DB 로그에 영구 저장되며, 후에 Rollback 연산이 Transaction 단위로 진행되도록 함

참고

	DML	DDL	DCL
Auto Commit 여부	X (SQL Server의 경우엔 O)	O (PostgreSQL의 경우엔 X)	O

- Auto Commit은 Rollback 불가능!

2) Rollback

- Transaction을 처음부터 다시 시작하거나, 부분적으로 연산된 결과를 취소하는 연산
- Transaction이 비정상 종료되어 원자성이 깨진 경우에 실행
- 사용자가 실행시킬 수 있음

1-4. Undo/Redo

1) Redo

- 복구

- 사용자가 했던 작업을 그대로 다시 진행 (Repeating History)
  - 가장 최근 검사점(Save Point)부터 나오는 모든 로그 레코드를 순회해 Redo하며 그 과정에서 `Undo-list` 생성

2) Undo

- 작업 롤백, 읽기 일관성, 복구
- 사용자가 했던 작업을 반대로 진행
  - Redo 단계에서 생성된 `Undo-list`를 역순회해 Undo함
  - 그 과정에서 수행한 연산을 CLR 형태로 로그에 기록

> 📖 **CLR(Compensation Log Record)**
>
> 해당 트랜잭션을 롤백하기 위해 기록하는 Undo 작업 로그

1-5. Schedule(History)

- 동시적으로 수행되는 여러 트랜잭션에 속하는 연산이 수행된 시간적 순서
- 동시적으로 수행된 트랜잭션의 모든 연산이 스케줄에 나와있어야 함
- 스케줄의 연산 순서는 해당 트랜잭션 내의 연산 순서와 동일해야 함

> 📖 **직렬 스케줄**
>
> 각 트랜잭션을 순서대로 시행한 스케줄
>
> 항상 올바르다. (오류가 생길 여지가 없다)

> 📖 **직렬 가능 스케줄(Serializable)**
>
> 직렬 스케줄과 동일한 결과를 내는 스케줄
>
> 스케줄 결과의 동일함을 정의하는 방식은 아래 두 가지가 있음
>
> - 충돌 직렬가능 스케줄 (Conflict Serializable)
> - 뷰 직렬가능 스케줄 (View Serializable)

0) 스케줄 예시

- Schedule 1
  - 직렬 스케줄
- Schedule 2
  - 직렬 스케줄

T1	T2
Read(A) A := A - 50 Write(A) Read(B) B := B + 50 Write(B)	Read(A) Temp := A * 0.1 Write(A) Read(B) B := B + Temp Write(B)

T1	T2
Read(A) A := A - 50 Write(A) Read(B) B := B + 50 Write(B)	Read(A) Temp := A * 0.1 Write(A) Read(B) B := B + Temp Write(B)

- Schedule 3
  - 직렬 스케줄은 아니지만, 위 스케줄과 결과가 동일
  - 직렬 가능 스케줄인지는 후에 서술

T1	T2
Read(A) A := A - 50 Write(A) Read(B) B := B + 50 Write(B)	Read(A) Temp := A * 0.1 Write(A) Read(B) B := B + Temp Write(B)

- Schedule 4
  - T1에서 A가 쓰여지기 전에 T2에서 A를 읽고 변경함.
  - 위 스케줄들과 결과가 다름. 올바른 스케줄이 아님

T1	T2
Read(A) A := A - 50 Write(A) Read(B) B := B + 50 Write(B)	Read(A) Temp := A * 0.1 Write(A) Read(B) B := B + Temp Write(B)

1) Conflict Serializable(충돌 직렬 가능 스케줄)

- 비충돌 연산을 바꾸면 직렬 스케줄이 되는 스케줄

📖 **충돌 연산(Conflict Instruction)**

순서를 바꾸면 결과가 달라지는 연산

연산 1	연산 2	충돌 여부
Read	Read	X
Read	Write	O
Write	Read	O
Write	Write	O

- ex1) 충돌 직렬 가능한 경우

T1	T2
Read(A) Write(A) Read(B) Write(B)	 Read(A) Write(A) Read(B) Write(B)

⇒

T1	T2
Read(A) Write(A) Read(B) Write(B)	 Read(A) Write(A) Read(B) Write(B)

- ex2) 충돌 직렬 가능하지 않은 경우

T1	T2
Read(Q) Write(Q)	 Write(Q)

2) View Serializable(뷰 직렬 가능 스케줄)

- 두 스케줄을 비교하며 직렬 가능성 판단
  1. 스케줄 S1의 각 트랜잭션의 읽기 연산에 대해 S2의 각 트랜잭션도 같은 값을 읽어야 함
  2. 스케줄 S1의 마지막 쓰기 연산이 S2의 마지막 쓰기 연산과 동일해야 함
- 보통 Blind Write 연산 존재 여부로 뷰 직렬가능 판단

📖 **Blind Write**

이전 값에 상관 없이 값을 덮어쓰는 것

- ex1) 뷰 직렬 가능한 경우
  - 충돌 직렬 가능하지 않지만, T3가 값을 덮어쓰게 되면서 직렬 스케줄과 결과가 같아짐

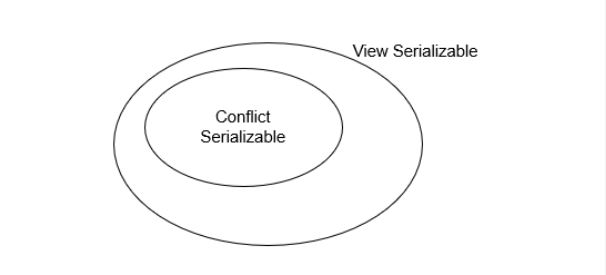
- S1

T1	T2	T3
Read(Q)		
Write(Q)	Write(Q)	
		Write(Q)

- S2(직렬 스케줄)

T1	T2	T3
Read(Q)		
Write(Q)		
	Write(Q)	
		Write(Q)

⚠ (충돌 직렬 가능) ⊂ (뷰 직렬 가능) 이다!



---

2. Isolation Level

- Transaction 격리 수준
- Transaction에서 일관성이 없는 데이터를 허용하는 수준
- 동시성 제어(Concurrency Control)를 위함

2-1. Isolation Level의 필요성

- DB는 ACID와 같이 원자적이면서 독립적인 수행을 함
- 따라서 `Locking`이라는 개념 등장
- 하지만 무조건적인 Locking으로 동시에 수행되는 많은 트랜잭션들을 순서대로 처리하면 DB 성능이 떨어짐
- 반대로 응답성을 높이기 위해 Locking 범위를 줄이면 잘못된 값이 처리될 가능성이 있음
- 따라서 최대한 효율적인 Locking 방법 필요

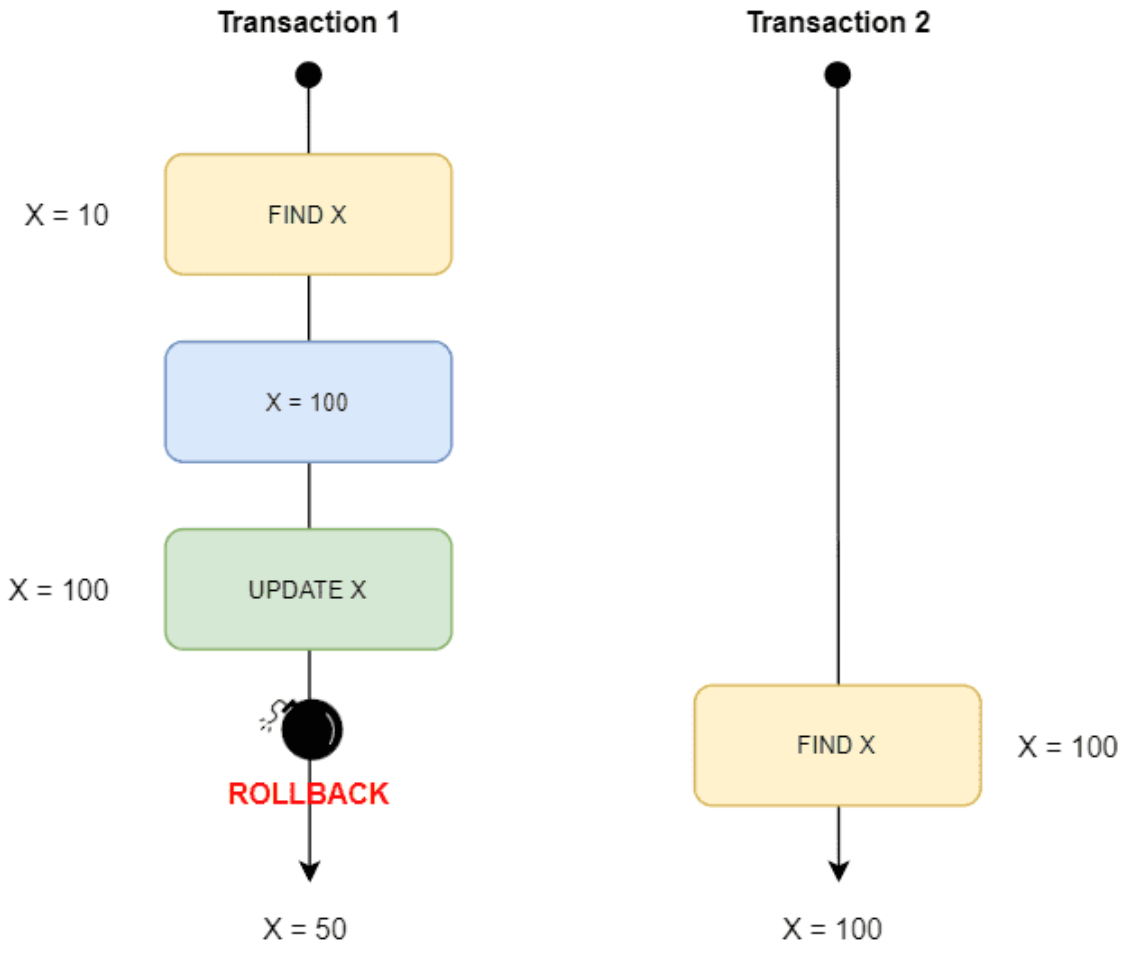
📖 **Locking**

Transaction이 DB를 다루는 동안 다른 Transaction이 관여하지 못하게 막는 것

2-2. Isolation Level의 종류

Isolation Level	Dirty Read	Non-repeatable Read	Phantom Read
Read Uncommitted	O	O	O
Read Committed	X	O	O
Repeatable Read	X	X	O
Serializable	X	X	X

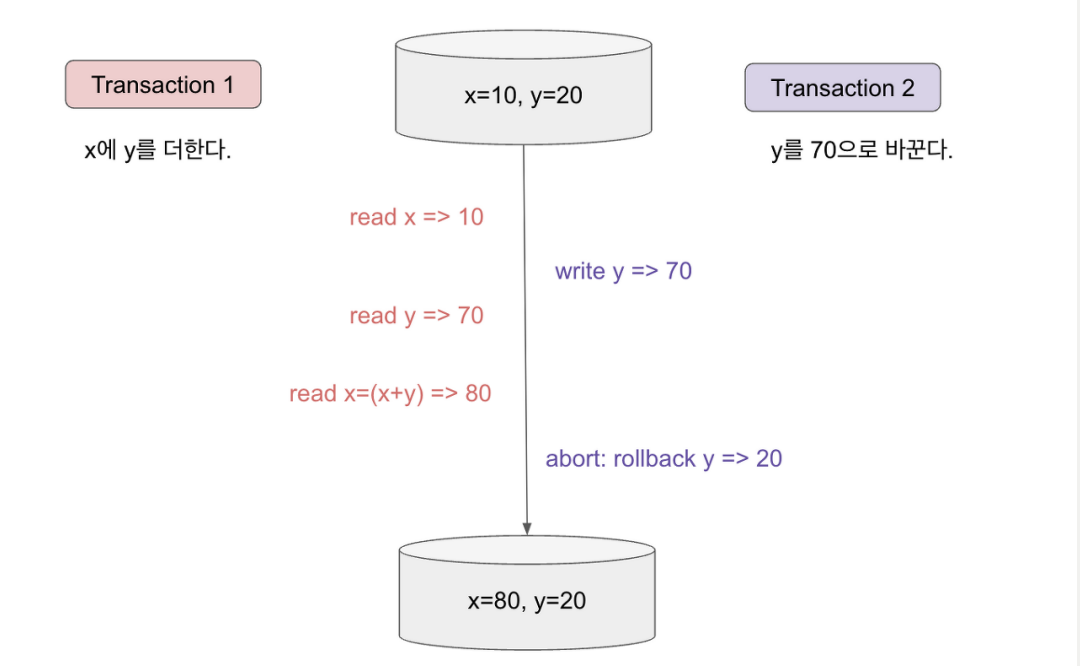
0) Level 0: Read Uncommitted



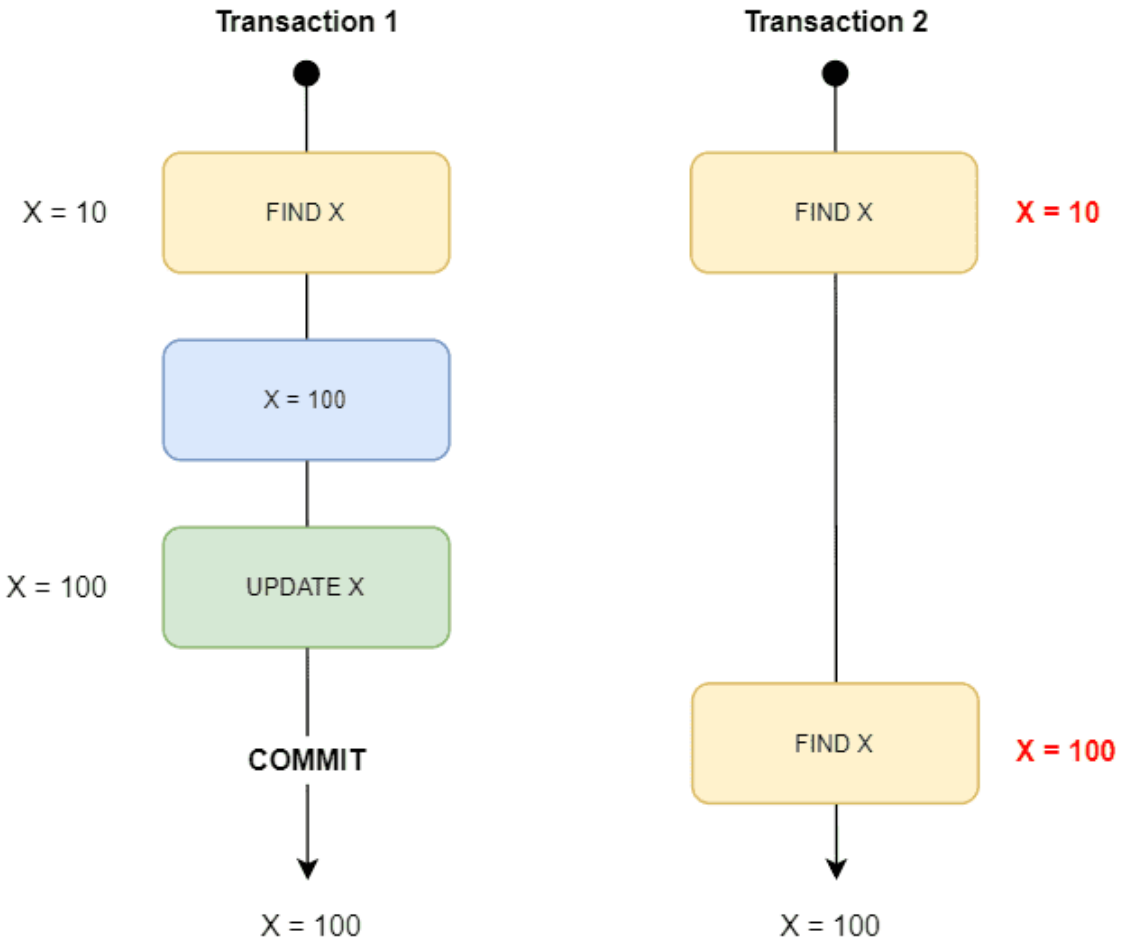
- `SELECT` **문장이 수행되는 동안 해당 데이터에 Shared Lock이 걸리지 않음**
- A Transaction 중 커밋되지 않은 데이터를 B Transaction에서 읽기 가능
- RDBMS 표준에서는 격리 수준으로 인정하지 않음
- `Dirty Read` 현상 발생
  - DB 일관성을 유지할 수 없음

📖 Dirty Read

Transaction이 완료되지 않았는데도 다른 Transaction에서 볼 수 있게 되는 현상



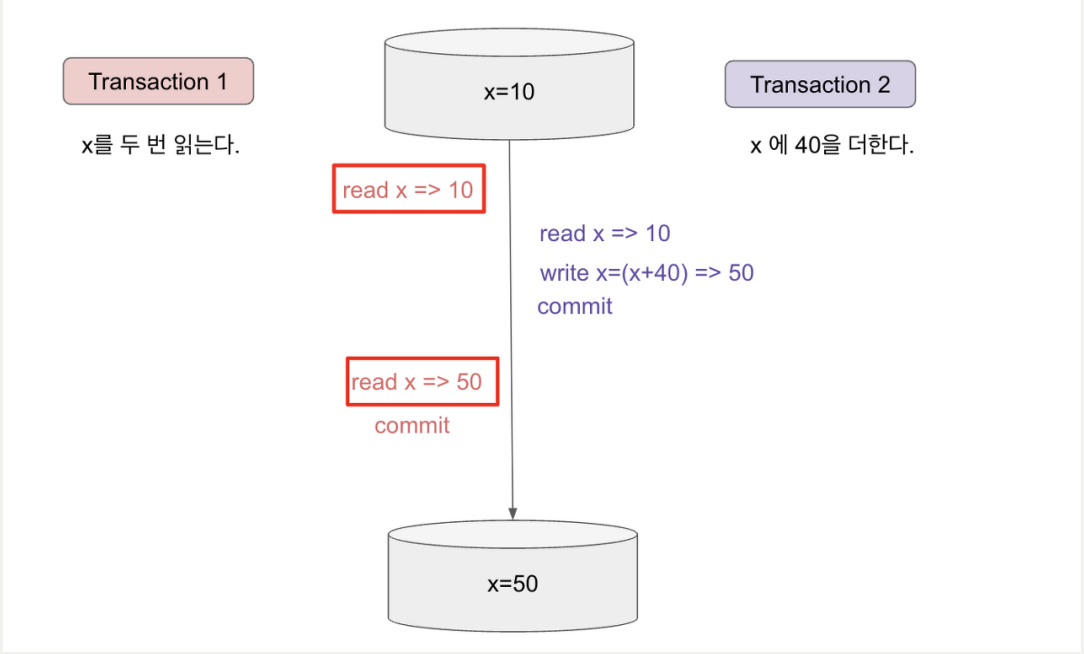
1) Level 1: Read Committed



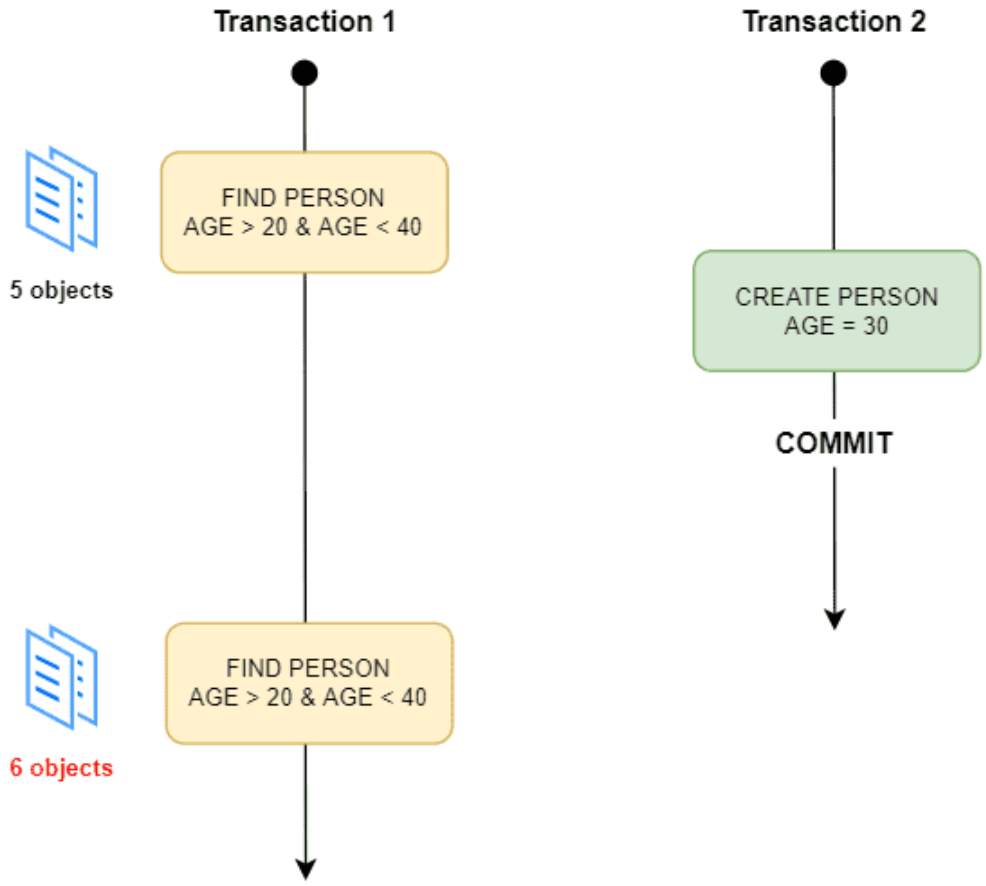
- `SELECT` **문이 실행되는 동안 Shared Lock이 걸림**
  - 조회 시에는 Undo 영역에 백업된 레코드값 가져옴
  - `Dirty Read` 방지
- 한 트랜잭션의 변경 내용이 Commit되어야만 다른 트랜잭션에서 조회 가능
- 대부분의 RDBMS에서 기본적으로 사용하는 격리 수준
- `Non-repeatable Read` 발생
  - DB 일관성을 유지할 수 없음

📖 Non-repeatable Read (Fuzzy Read)

하나의 트랜잭션에서 같은 SELECT 쿼리를 여러번 실행했을 때 다른 결과가 나오는 경우



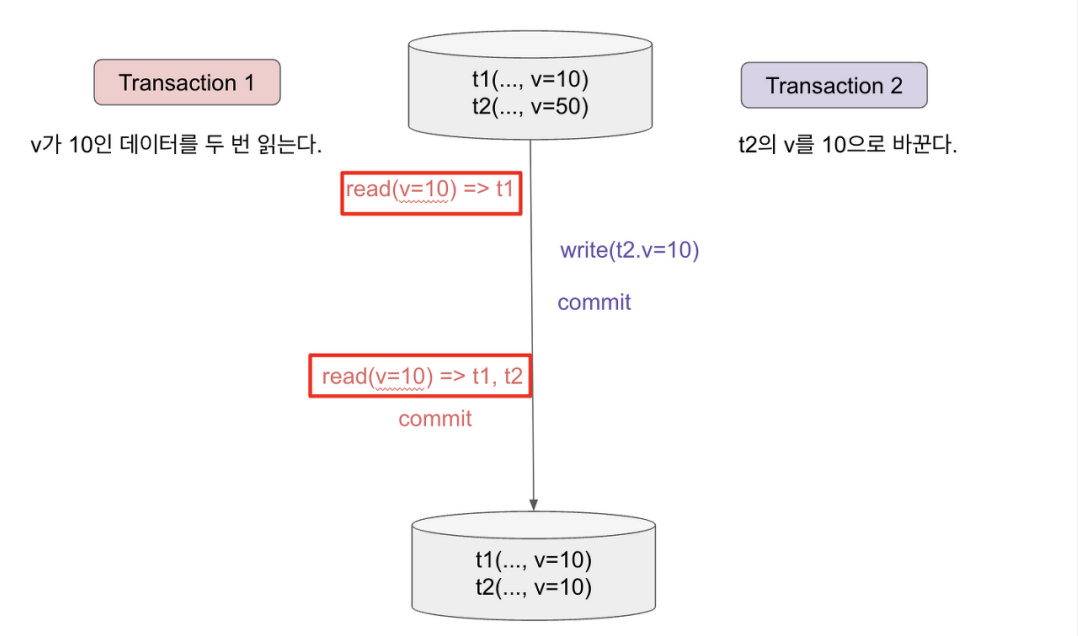
2) Level 2: Repeatable Read



- **<u>트랜잭션이 완료될 때 까지</u>** `SELECT` **문이 사용하는 모든 데이터에 Shared Lock이 걸림**
  - `Non-repeatable Read` 방지
- 트랜잭션이 시작되기 전에 커밋된 내용에 대해서만 조회 가능
- MySQL에서 기본으로 사용하는 격리수준
- 트랜잭션 시작 시점 데이터의 일관성을 보장해야 하므로 실행 시간이 길어질 수록 **멀티 버전을 관리**해야 하는 단점 발생
- `Phantom Read` 발생
  - 다른 트랜잭션에 의해 테이블의 데이터 자체가 INSERT/DELETE될 수 있음

📖 Phantom Read

한 트랜잭션에서 같은 조건으로 테이블을 2번 읽었는데, 2번의 결과가 다른 현상.



💡 PostgreSQL에서는 MVCC(Multi-version Concurrency Control)를 사용하여 특정 버전 시점의 DB를 조회할 수 있으므로 행이 추가/삭제된 내역도 반영되지 않아 Repeatable Reade 단계에서 Phantom Read가 발생하지 않음

다른 단계에서 발생시 SSI(Serializable Snapshot Isolation)으로 해결

3) Level 3: Serializable

- **트랜잭션들이 동시에 일어나지 않고, 순차적으로 실행되는 것 처럼 동작**
  - `Phantom Read` 방지
- 가장 단순하면서 엄격한 격리 수준
- 처리 성능이 가장 낮음
- 거의 사용되지 않음

+) Snapshot Isolation

- **트랜잭션 시작 이전에 Commit된 데이터만 보임**
  - Phantom Read 방지
- 동시성 제어가 어떻게 동작할 지 구현을 바탕으로 정의된 Isolation Level

- MVTO(Multi-version Timestamp Ordering) 기법 사용
  - MVCC(Multi-version Concurrency Control)의 한 종류
  - date+time 형식의 타임스탬프를 사용하여 관리
  - 각 데이터 버전은 [버전의 값, 읽기 타임스탬프, 쓰기 타임스탬프]를 가짐
    - 읽기 타임스탬프: 해당 값을 읽은 가장 최근 트랜잭션의 타임스탬프
    - 쓰기 타임스탬프: 해당 버전에 새로운 값을 쓴 트랜잭션의 타임스탬프
  - 읽기연산은 항상 기다림 없이 진행
    - 읽기 연산에 대한 데드락 상태 방지
- Oracle, PostgreSQL, SQL Server 등의 시스템 및 오픈소스 시스템에서 폭넓게 사용됨
- 트랜잭션 충돌에 대해서는 아래 두 방법 사용
  - **첫번째 커밋승(First-committer-wins)**: 나중에 커밋한 트랜잭션이 롤백됨
  - **첫번째 갱신승(First-updater-wins)**: 나중에 업데이트한 트랜잭션이 롤백됨