Transaction

💬 Transaction이란?

: 트랜잭션은 DB에서 매우 중요한 개념 중 하나로 DBMS에서 데이터를 다루는 논리적인 작업의 단위 이다. 하나의 트랜잭션은 여러 개의 데이터 연산으로 구성되어 있다.
구체적으론, 여러 개의 연산이라 할 수도 있고, SQL에서 하나의 연산을 두 DB의 relation에 접근해서 양쪽에 동일하게 반영시킬 때도 트랜잭션으로 묶어주게 된다.
일반적으로는 하나의 SQL문을 하나의 트랜잭션으로 보지만 실제로는 여러 개의 SQL문으로 이루어진 트랜잭션도 존재한다.

💬 Transaction의 목적/ 용도

: DB에서 이런 트랜잭션을 정의하는 이유는 **데이터 작업 중 장애가 발생할 때 데이터를 복구하는 작업의 단위**로 사용하기 위함이다.
데이터 작업 중 장애가 발생해서 트랜잭션을 제대로 수행하지 못했다면 그 트랜잭션이 일어나기 전 상태로 보류시켜야 하는데, 이 복구 시점을 어디까지로 잡을 것이냐에 대한 기준을 트랜잭션으로 잡는 것이다.
또한,
여러 작업이 동시에 같은 데이터를 다룰 때 작업을 서로 분리하기 위한 단위로 사용되기 위함도 있다.

💬 Transaction의 성질

- **원자성 (Atomicity)**

  : **all or nothing**을 의미한다. 하나의 트랜잭션에 포함된 작업은 전부 수행되거나 전혀 수행되지 않아야 한다. 트랜잭션 수행 도중 오류가 발생하면, 일부만 수행되는 일이 없도록 DBMS의 회복 모듈을 사용하여 원자성을 보장한다.

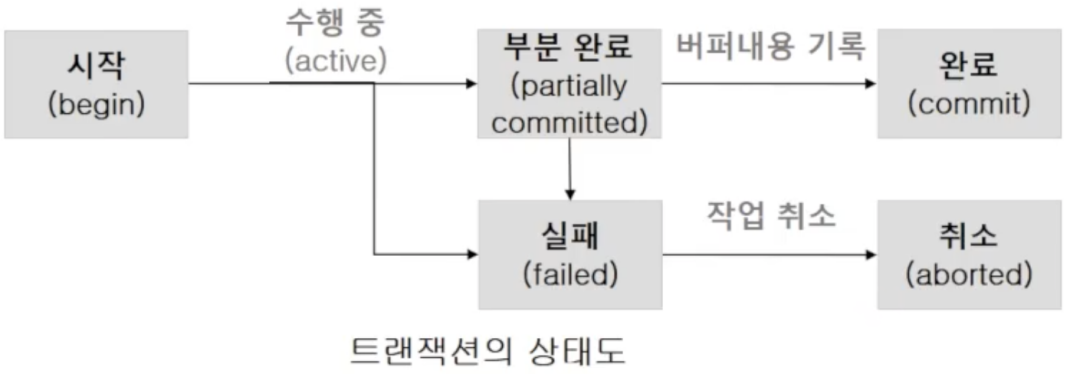
- **일관성 (Consistency)**
  :
  기본적으로 데이터들은 항상 일관성을 가져야 한다. 다만, 트랜잭션 도중에는 일시적으로 일관되지 않을 수 있다. 트랜잭션이 수행되기 전에도 일관적인 상태를 가져야 하고, 트랜잭션 후에도 새로운 일관된 상태를 유지해야 한다.

- **고립성 (Isolation)**

  : 한 트랜잭션이 데이터를 갱신하는 동안 다른 트랜잭션이 이 데이터에 접근하지 못하게 막아야 한다. 다수의 트랜잭션이 동시에 수행되더라도 그 결과가 순차적으로 수행된 것과 같음을 보장하는 **DBMS의 동시성 제어 기능**이 고립성을 보장한다.

- **지속성 (Durability)**
  : 성공적으로 완료된 트랜잭션은 변경한 데이터를 영구히 저장해야 한다. 한번 commit이 되면 그 데이터는 DB에

영구적으로 저장되어야 한다.



트랜잭션의 상태도

- **활성 (Active)** : 트랜잭션이 정상적으로 실행중인 상태

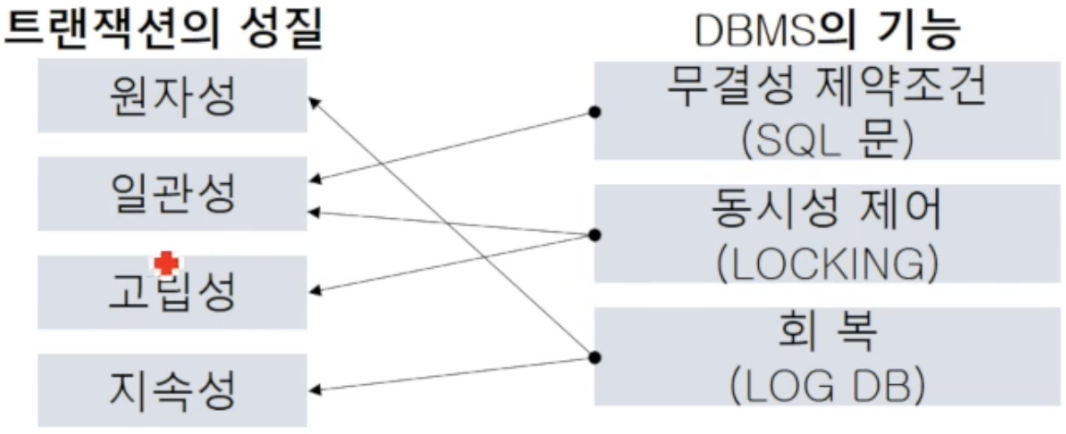
- 작업 성공 시,

- **부분 완료 (Partially Commited)** : 트랜잭션의 마지막까지 실행되었지만, Commit 연산이 실행되기 직전의 상태
- **완료 (Commited)** : 트랜잭션 성공이 종료되어 Commit 연산을 실행한 후의 상태

- 작업 실패 시,

- **실패 (Failed)** : 트랜잭션 실행에 오류가 발생하여 중단된 상태
- **철회 (Aborted)** : 트랜잭션이 비정상적으로 종료되어 Rollback 연산을 수행한 상태

💬 Transaction의 성질과 DBMS의 기능



트랜잭션의 성질과 DBMS의 기능

1. **무결성 제약 조건** : 어떤 권한을 가진 사용자가 실수로 데이터 일관성을 헤치는 것을 막기 위해 존재한다.
   (ex. 은행 잔고는 양수여야 한다, 학생 ID는 기본 키다.)

2. **회복** : 트랜잭션이 데이터를 갱신하는 도중에 시스템이 고장나도 데이터 일관성 유지를 위해 복구 시켜주는 기능이다.

3. **동시성 제어** : 대부분의 DBMS는 다수의 사용자를 가지므로, 동일한 테이블에 여러 사용자가 동시에 접근하기도 한다.
   데이터 일관성 유지를 위해서라면 동시 접근 자체를 막으면 되겠지만, 성능을 고려하면 가능한 한 동시 접근을 허용하는 것이 필수적이다. 동시성 제어 기법은
   여러 사용자들이 다수의 트랜잭션을 동시에 수행하는 환경에서도, 트랜잭션들 간 간섭이 생기지 않도록 하여 일관성을 보장하는 기능이다.

   동시성 제어를 하는 방법에는 대표적으로 "Lock-based 프로토콜"과 "Timestamp-based 동시성 제어가 있다.

   - **방법 1 : Lock-based 프로토콜**

     널리 사용되는 locking 개념을 사용한 방법이다. 어떤 트랜잭션이 해당 데이터 갱신을 목적으로 접근할 때는 `독점락 (X-lock, exclusive lock)`을 요청하고, 읽기를 목적으로 접근할 때는 `공유락 (S-lock, Shared lock)`을 요청한다.
     공유락끼리는 서로의 공존을 허용하지만, 독점락은 다른 어떤 락과도 공존을 허용하지 않는다.

		현재 걸려 있는 락		
		공유 락	독점 락	락이 걸려 있지 않음
요청중인 락	공유 락	허용	대기	허용
	독점 락	대기	대기	허용

     여기서 Lock 정보는 DB 내 각 데이터 항목과 연관된 하나의 변수이며, Lock 테이블 등에 유지된다.

   - **방법 2 : Timestamp-based 동시성 제어**

     여기서는 락을 걸지 않고 타임 스탬프를 사용하여 트랜잭션을 안전하게 다룬다. 모든 트랜잭션은 시작할 때 고유한 타임스탬프를 가지며, 나중에 시작한 트랜잭션이 더 타임스탬프 값을 가진다.
     두 트랜잭션 내 작업들이 충돌했을 때 이 타임스탬프를 기준으로 어떤 작업이 먼저 수행되어야 하는지 판단한다.
     DB의 각 데이터는 각각
     `RTS (Read timestamp)`와 `WTS (Write timestamp)` 를 가진다.

RTS는 해당 데이터를 누군가 읽을 때마다 갱신 되고, WTS는 쓸 때마다 갱신 된다.

```
어떤 트랜잭션 Ti가 데이터 D를 읽겠다고(Read) 요청하면, Ti의 스탬프와 D에 적힌 WTS를 비교한다.
1. Ti의 스탬프가 D의 WTS보다 크거나 같다면(=읽기가 마지막 쓰기보다 나중이라면) 작업을 허용한다.
2. Ti의 스탬프가 D의 WTS보다 작다면 읽기 이후에 데이터가 수정된 것이므로 거부한다. 작업이 끝나면 해당 데이터의 RTS를 업데이트 한다.

어떤 트랜잭션 Ti가 데이터 D를 쓰겠다고(Write) 요청하면, Ti의 스탬프와 D에 적힌 RTS/WTS를 모두 확인한다.
1. Ti가 RTS보다 작다면 누군가 미래 시점에 이미 데이터를 읽어버렸으므로 갱신할 수 없어 거부한다.
2. Ti가 WTS보다 작다면 누군가 미래 시점에 이미 데이터를 갱신해버렸으므로 거부하고, Ti가 했던 작업을 전부 롤백한다.
3. 그 외 경우에는 쓰기 작업이 가능하다.
```

💬 스프링의 트랜잭션 전파속성(Transaction propagation)

- **트랜잭션의 시작과 종료**
  : 트랜잭션은 시작 지점과 끝나는 지점이 존재한다. 시작하는 방법은 1가지이지만, 끝나는 방법은 2가지이다. 트랜잭션이 끝나는 방법에는 모든 작업을 확정 짓는 `커밋(commit)`과 모든 작업을 무효화하는 `롤백(rollback)`이 있다.

- **트랜잭션의 시작**

  : 트랜잭션은 **하나의 Connection을 가져와 사용하다가 닫는 사이**에서 일어난다.

  트랜잭션의 시작과 종료는 Connection 객체를 통해 이뤄지기 때문이다.
  JDBC의 기본 설정은 DB 작업을 수행한 직후에 바로 커밋을 하는 자동 커밋 옵션이 활성화되어 있다. 그러므로 JDBC에서 트랜잭션을 시작하려면 자동 커밋 옵션을 false로 해주어야 하는데, 그러면 새로운 트랜잭션이 시작되게 만들 수 있다.

```
public void executeQuery throws SQLException {
        Connection connection = dataSource.getConnection();
        connection.setAutoCommit(false);
        // 트랜잭션 시작

}
```

스프링을 이용하면 내부적으로 커넥션을 갖고 있는 **추상화된 트랜잭션 매니저**를 이용하게 된다. 다음과 같이 트랜잭션을 시작하게 되고, 자동 커밋 옵션을 변경하는 등의 작업은 트랜잭션 매니저 내부에서 진행된다.

```
public void executeQuery() throws SQLException {
        TransactionStatus status = transactionManager.getTransaction(new
DefaultTransactionDefinition());
        // 트랜잭션 시작

}
```

- **트랜잭션의 종료**

  : 하나의 트랜잭션이 시작되면 **commit() 또는 rollback() 호출될 때까지가 하나의 트랜잭션**으로 묶인다.
  이렇게
  `setAutoCommit(false)` 로 트랜잭션의 시작을 선언하고, `commit()` 또는 `rollback()` 으로 트랜잭션을 종료하는 작업을 `트랜잭션의 경계 설정`이라고 한다.
  트랜잭션의 경계는 하나의 Connection을 통해 진행되므로 트랜잭션의 경계는 하나의 커넥션이 만들어지고 닫히는 범위 안에 존재한다.

```
public void executeQuery() throws SQLException {
        TransactionStatus status = transactionManager.getTransaction(new
DefaultTransactionDefinition());
// 트랜잭션 시작

        try{
            // 쿼리 실행
                transactionManager.commit(status);
        } catch (Exception e){
                transactionManager.rollback(status);
        }
}
```

- **트랜잭션 전파 속성(Transaction Propagation)이란?**

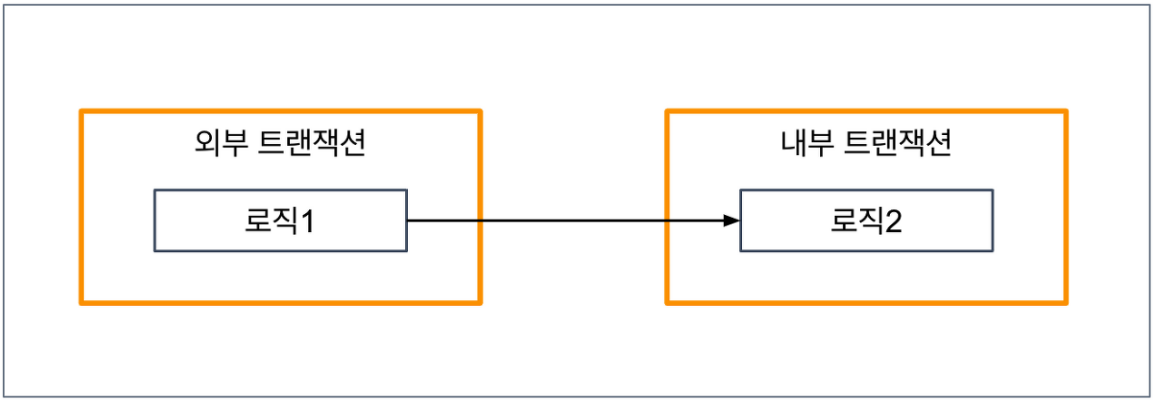
  : Spring이 제공하는 선언적 트랜잭션 (트랜잭션 어노테이션, `@Transactionial`)의 장점 중 하나는 여러 트랜잭션을 묶어서 하나의 트랜잭션 경계를 만들 수 있다는 점이다.

  작업을 하다보면 기존에 트랜잭션이 진행중일 때 추가적인 트랜잭션을 진행해야 하는 경우가 있다.

  이미 트랜잭션이 진행중일 때 추가적인 진행을 어떻게 할지 결정하는 것이 `전파 속성(Propagation)`이다.

  전파 속성에 따라 기존의 트랜잭션에 참여할 수도 있고, 별도의 트랜잭션으로 진행할 수도 있고, 에러를 발생시키는 등 여러 선택을 할 수 있다.

아래는 하나의 트랜잭션이 다른 트랜잭션을 만나는 상황을 그림으로 나타낸 것이다.

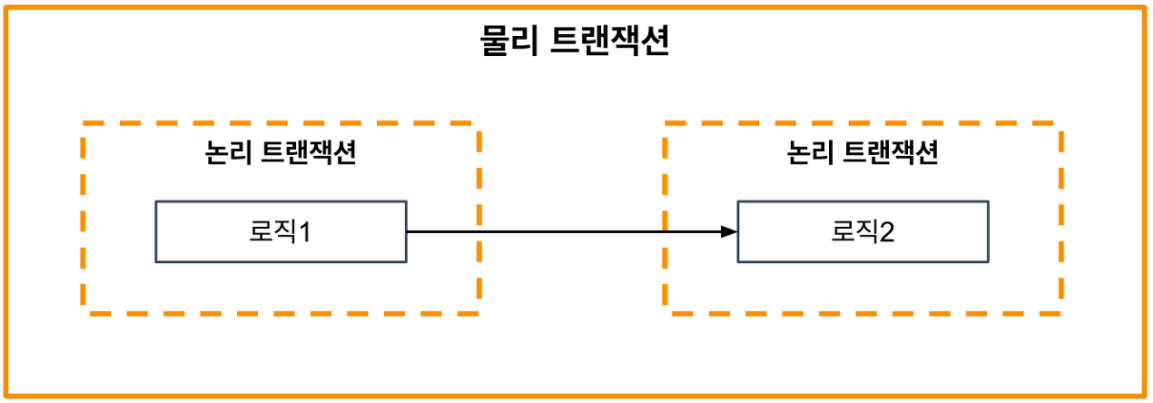


- **물리 트랜잭션과 논리 트랜잭션**

: 트랜잭션은 데이터베이스에서 제공하는 기술이므로 **커넥션 객체를 통해 처리**한다. 그래서 **1개의 트랜잭션을 사용한다는 것은 하나의 커넥션 객체를 사용한다는 것**이고, 실제 데이터베이스의 트랜잭션을 사용한다는 점에서 `물리 트랜잭션`이라고도 한다.

앞서 설명하였듯이, 트랜잭션 전파 속성에 따라 외부 트랜잭션과 내부 트랜잭션이 동일한 트랜잭션을 사용할 수도 있다. 하지만 스프링의 입장에서는 트랜잭션 매니저를 통해 트랜잭션을 처리하는 곳이 2군데이다.

그래서 실제 데이터베이스 트랜잭션과 스프링이 처리하는 트랜잭션 영역을 구분하기 위해 스프링은 `논리 트랜잭션`이라는 개념을 추가하였다. 예를 들어 다음 그림은 외부 트랜잭션과 내부 트랜잭션이 1개의 물리 트랜잭션을 사용하는 경우이다.



이 경우에는 2개의 트랜잭션 범위가 존재하기 때문에 개별 논리 트랜잭션이 존재하지만, 실제로는 1개의 물리 트랜잭션이 사용된다.
만약 트랜잭션 전파 없이 1개의 트랜잭션만 사용되면 물리 트랜잭션만 존재하고, 트랜잭션 전파가 사용될 때는 논리 트랜잭션 개념이 사용된다.

- **물리 트랜잭션** : 실제 데이터베이스에 적용되는 트랜잭션으로, 커넥션을 통해 커밋/롤백하는 단위
- **논리 트랜잭션** : 스프링이 트랜잭션 매니저를 통해 트랜잭션을 처리하는 단위

기존의 트랜잭션이 진행중일 때, 또 다른 트랜잭션이 사용되면 복잡한 상황이 발생한다. 스프링은 논리 트랜잭션이라는 개념을 도입함으로써 상황에 대한 설명을 쉽게 만들고, 다음과 같은 단순한 원칙을 세울 수 있었다.

- 모든 논리 트랜잭션이 커밋되어야 물리 트랜잭션이 커밋됨
- 하나의 논리 트랜잭션이라도 롤백되면 물리 트랜잭션을 롤백됨

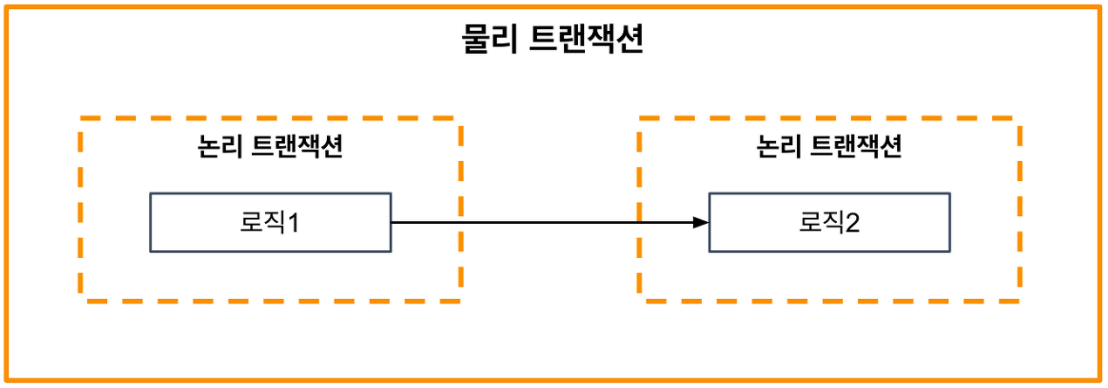
💬 다양한 스프링의 Transaction 전파 속성

- **REQUIRED 속성과 REQUIRES_NEW 속성**

  : **스프링에는 7가지 전파 속성**이 존재하는데, REQUIRED와 REQURIES_NEW를 바탕으로 어떻게 진행되는지 살펴보자.

  - **REQUIRED** `@Transactional(propagation = Propagation.REQUIRED)`

    : REQUIRED는 스프링이 제공하는 **기본 옵션**으로, 기본적으로 **2개의 논리 트랜잭션을 묶어 1개의 물리 트랜잭션**을 사용하는 것이다. 앞선 예시로 살펴본 경우가 REQUIRED에 해당하며, 내부 트랜잭션은 기존에 존재하는 외부 트랜잭션에 참여하게 된다.



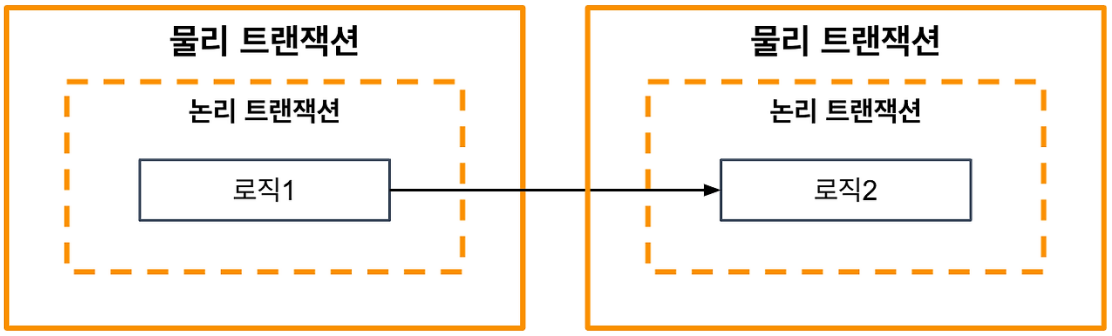
여기서 참여한다는 것은 외부 트랜잭션을 그대로 이어나간다는 뜻이며, 외부 트랜잭션의 범위가 내부까지 확장되는 것이다. 그러므로 내부 트랜잭션은 새로운 물리 트랜잭션을 사용하지 않는다.

하지만 트랜잭션 매니저에 의해 관리되는 논리 트랜잭션이 존재하므로, 커밋은 내부 1회, 외부 1회해서 총 2회 실행된다.
물론 내부 트랜잭션은 논리 트랜잭션이기 때문에 커밋을 호출해도 즉시 커밋되지는 않고, 외부 트랜잭션이 최종적으로 커밋될 때 실제로 커밋이 된다.

롤백 역시 비슷한데, 내부 트랜잭션에서 롤백을 하여도 즉시 롤백되지 않는다. 물리 트랜잭션이 롤백될 때 실제 롤백이 처리되는데, 논리 트랜잭션들 중 1개라도 롤백되었다면 롤백된다.
물리 트랜잭션은 실제 커넥션에 롤백/ 커밋을 호출하는 것이므로 해당 트랜잭션이 끝나는 것이다.

- **REQUIRES_NEW** `@Transactional(propagation = Propagation.REQUIRES_NEW)`

  : REQUIRES_NEW는 외부 트랜잭션과 내부 트랜잭션을 완전히 분리하는 전파 속성이다. 그래서 2개의 물리 트랜잭션이 사용되며, 각각 트랜잭션 별로 커밋과 롤백이 수행된다.



두 개는 서로 다른 물리 트랜잭션이므로, 내부 트랜잭션 롤백이 외부 트랜잭션 롤백에 영향을 주지 않는다. 그러므로 내부 트랜잭션이 롤백 호출은 실제 커넥션에 롤백을 호출하는 것이므로 트랜잭션이 끝나게 된다.

서로 다른 물리 트랜잭션을 가진다는 것은 **각각의 DB 커넥션**이 사용된다는 것이다.
즉, 1개의 HTTP 요청에 대해 2개의 커넥션이 사용되는 것이다. 내부 트랜잭션이 처리 중일때는 꺼내진 외부 트랜잭션이 대기하는데, 이는 데이터베이스 커넥션을 고갈시킬 수 있다. 그러므로 조심해서 사용해야 하며, REQUIRE_NEW없이 해결 가능하다면 대안책 (별도의 클래스 두기 등)을 사용하는 것이 좋다.

(ex. 회원 가입과 이력 로그를 저장하는 로직을 하나의 트랜잭션으로 묶어서 처리하면, 이력 로그 저장에서 문제가 발생하면 회원가입 자체가 안되는 상황이 발생한다.

→
회원 가입을 시도한 로그를 남기는데 실패하더라도 회원 가입은 유지되게 해줄 때 REQUIRES_NEW를 사용할 수 있다.

이를 통해 이력 로그에 에러가 발생해도 회원 데이터는 저장되고, 회원 로그 데이터만 롤백되는 것을 확인할 수 있다. 하지만, 하나의 HTTP 요청에 2개의 데이터 베이스 커넥션을 사용하게 되므로, 성능이 중요한 곳에서는 주의가 필요하다.

- 그 외 트랜잭션의 전파 속성 요약

	의미	기존 트랜잭션 X	기존 트랜잭션 O
REQUIRED	트랜잭션이 필요함 (없으면 새로 만듬)	새로운 트랜잭션을 생성함	기존 트랜잭션에 참여함
SUPPORTS	트랜잭션이 있으면 지원함 (트랜잭션이 없어도 됨)	트랜잭션 없이 진행함	기존 트랜잭션에 참여함
MANDATORY	트랜잭션이 의무임 (트랜잭션이 반드시 필요함)	IllegalTransactionStateException 예외 발생	기존 트랜잭션에 참여함
REQUIRES_NEW	항상 새로운 트랜잭션이 필요함	새로운 트랜잭션을 생성함	기존 트랜잭션을 보류시키고 새로운 트랜잭션을 생성함
NOT_SUPPORTED	트랜잭션을 지원하지 않음 (트랜잭션 없이 진행함)	트랜잭션 없이 진행함	기존 트랜잭션을 보류시키고 트랜잭션 없이 진행함
NEVER	트랜잭션을 사용하지 않음 (기존 트랜잭션도 허용하지 않음)	트랜잭션 없이 진행	IllegalTransactionStateException 예외 발생
NESTED	중첩(자식) 트랜잭션을 만듬	새로운 트랜잭션을 생성함	중첩 트랜잭션을 만듬

💬 Transaction의 격리 수준

- **격리 수준 (isoaltion level) 이란?**

  : 격리 수준 (isolation level)이란 트랜잭션끼리 **얼마나 서로 고립되어 있는지를 나타내는 수준**이다.
  즉, 한 트랜잭션이
  다른 트랜잭션이 변경한 데이터에 대한 접근 강도를 의미한다.
  레벨이 높아질수록 트랜잭션간 고립정도가 높아지며, 성능 저하도 야기된다.
  일반적인 온라인 서비스에서는
  `READ COMMITED`나 `REPEATABLE READ` 중 하나를 사용한다.

	DIRTY READ	NON-REPEATABLE READ	PHANTOM READ
READ UNCOMMITTED	발생	발생	발생
READ COMMITTED	발생하지 않음	발생	발생
REPEATABLE READ	발생하지 않음	발생하지 않음	발생
SERIALIZABLE	발생하지 않음	발생하지 않음	발생하지 않음

- **격리 수준의 필요성**

  : 격리수준은 트랜잭션은 `ACID 특성`을 보장하기 위해 사용한다.

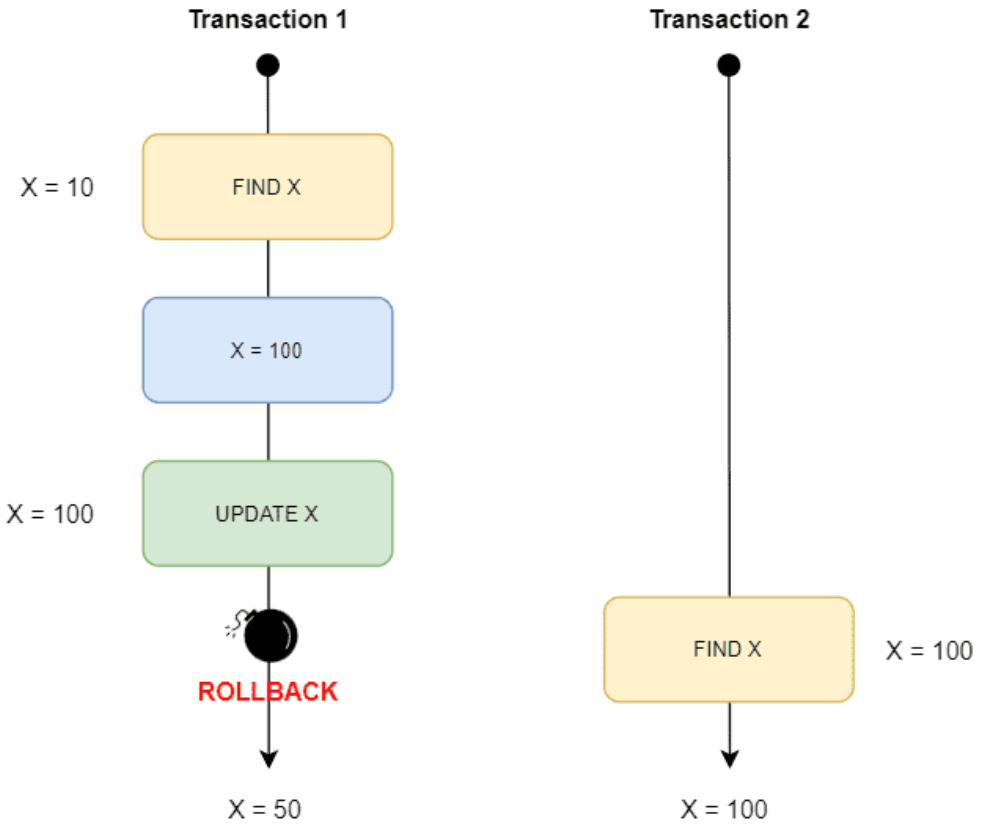
  `Locking`을 통해 이를 해결할 수 있지만, 무조건적인 Locking은 성능 저하를 가져온다. 반대로 느슨한 Locking은 데이터 무결성에 큰 문제를 가져온다.
  효율적인 Locking의 사용을 위해
  적절한 격리수준은 반드시 필요하다.

- **상호작용**
  - `DirtyRead` : 아직 **커밋(Commit)되지 않은 다른 트랜잭션의 데이터**를 읽는 것을 말한다.
  - `Non-repeatable Read` : 다른 트랜잭션이 커밋(Commit)한 데이터를 읽을 수 있는 것을 의미한다.
    즉, 한 트랜잭션에서 같은 쿼리로 2번 이상 조회했을 때,
    그 결과가 상이한 상황을 말한다.
    보통
    데이터의 수정/삭제가 발생했을 경우 발생한다.
  - `Phantom Read` : 다른 트랜잭션이 커밋(Commit)한 데이터가 있더라도 자신의 트랜잭션에서 읽었던 내용만 사용하는 것을 의미한다.
    즉, 한 트랜잭션에서 같은 쿼리를 2번 이상 조회했을 때
    없던 결과가 조회되는 상황을 말한다.
    보통
    데이터의 삽입이 발생했을 경우 발생한다.

- **격리 수준**
  1. Read Uncommmited (Level 0)

  : 어떤 트랜잭션의 내용이 커밋(Commit)이나 롤백(Rollback)과 상관없이 다른 트랜잭션에서 조회가 가능하다.
  정합성의 문제가 많은 격리 수준이기 때문에 RDBMS 표준에서는 **격리수준으로 인정하지 않는다**.

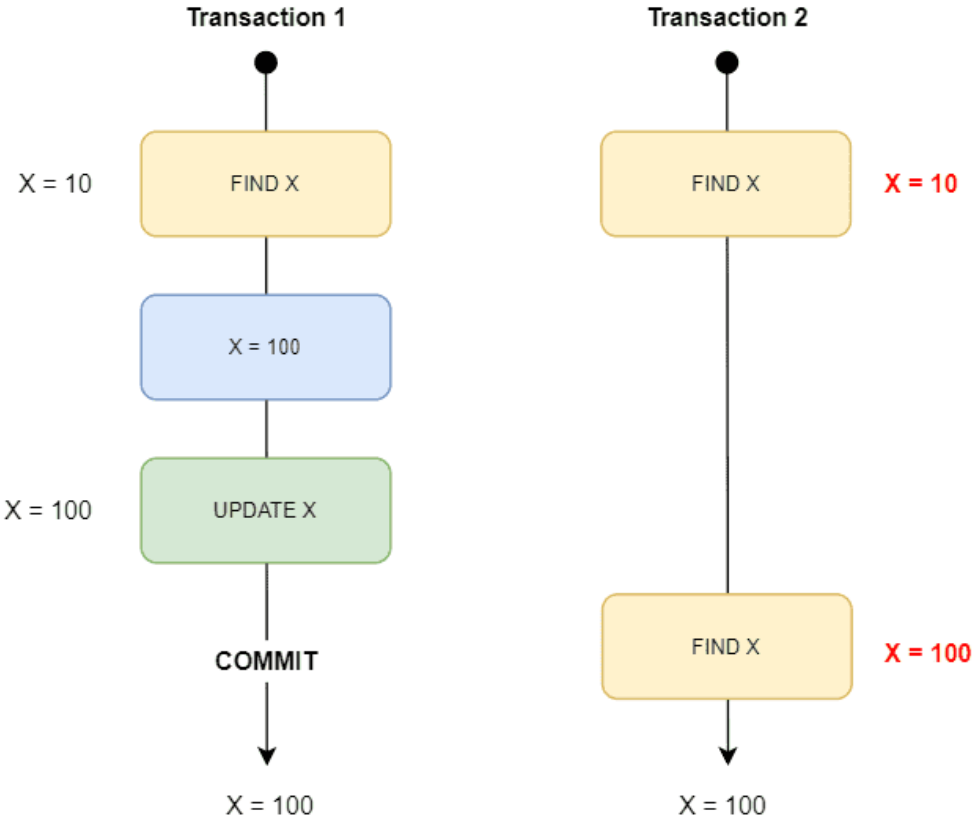
  Select 문이 실행되는 동안 해당 데이터에 **Shared Lock이 걸리지 않아**, `Dirty Read`가 발생한다.



2. **Read Commited (Level 1)**

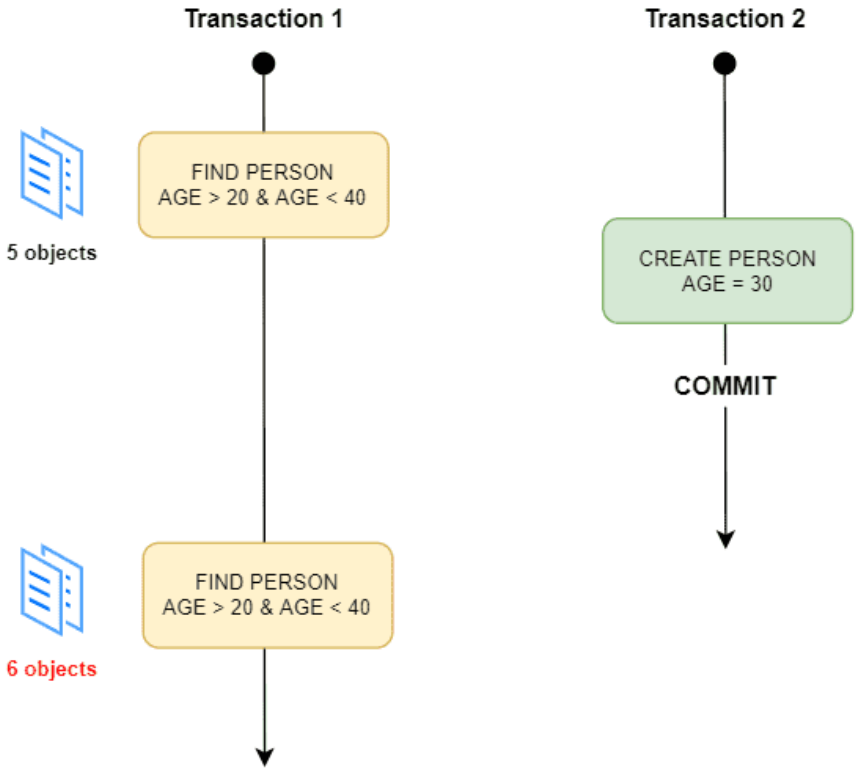
: 한 트랜잭션의 변경 내용이 **커밋(Commit) 되어야만** 다른 트랜잭션에서 조회가 가능하다. 대부분의 RDBMS에서 **기본적으로 사용하는 격리수준**이다.
Select 문이 실행되는 동안 Shared Lock이 걸린다. 조회 시에는 실제 테이블 값이 아니라
Undo 영역에 백업된 레코드 값을 가져온다.

하지만 하나의 트랜잭션에서 똑같은 SELECT 쿼리를 실행했을 때는 항상 같은 결과를 가져와야 하는 `REPEATABLE READ`의 정합성에 어긋난다. 즉, `Non-repeatable Read`가 발생한다.



3. **Repeatable Read (Level 2)**

: 트랜잭션이 시작되기 전에 커밋된 내용에 대해서만 조회가 가능하다. MySQL에서 기본적으로 사용하며, 이 격리 수준에서는 `Non-repeatable Read`가 발생하지 않는다.
트랜잭션이 완료될 때까지 Select 문이 사용하는 모든 데이터에 Shared Lock이 걸린다. 따라서 트랜잭션이 범위 내에서 조회한 데이터의 내용이 항상 동일함을 보장한다.
트랜잭션이 시작 시점 데이터의 일관성을 보장해야 하기 때문에 트랜잭션의 실행시간이 길어질수록 계속 **멀티버전을 관리해야 하는 단점**이 있다.
하지만,
`Phantom Read`가 발생할 수 있다.



4. **SERIALIZABLE (LEVEL3)**

: 가장 단순하면서 엄격한 관리 수준이지만, 성능 측면에서는 **동시 처리 성능이 가장 낮다.** SERIALIZABLE에서는 `PHANTOM READ`가 발생하지 않는다.

트랜잭션들이 동시에 일어나지 않고, 순차적으로 실행되는 것처럼 동작한다.

거의 사용되지 않는다.

< 참고자료 >

[Spring] 스프링의 트랜잭션 전파 속성(Transaction propagation) 완벽하게 이해하기

아래의 내용은 김영한님의 디비 접근 기술 2편 강의와 토비의 스프링 등을 바탕으로 정리한 내용입니다. 1. 트랜잭션의 시작과 종료 및 전파 속성(Transaction Propagation) [트랜잭션의 시작과 종료] 트랜잭션은 시작 지점과 끝나는 지점이 존재한다. 시작하는 방법은 1가지이지만 끝나는 방법은

https://mangkyu.tistory.com/269

	의미	기존 트랜잭션 X	기존 트랜잭션 O
REQUIRED			
SUPPORTS			
MANDATORY			
REQUIRES_NEW			
NOT_SUPPORTED			
NEVER			
NESTED			

[DB 기술면접 질문 리스트] : 인덱스

- RDBMS가 뭔가요? RDBMS의 특징 설명해보시겠어요? 관계형 데이터베이스를 생성하고 수정하고 관리할 수 있는 소프트웨어입니다. RDBMS의 특징으로는 모든 데이터를 2차원 테이블로 표현하며, 각 테이블의 단위는 row(record, tuple)과 column(field, item)으로 이루어져있습니

https://land-turtler.tistory.com/120

클러스터드 인덱스	넌클러스터드 인덱스
행을 물리적으로 재배열 한다.	행을 물리적으로 배열 하
인덱스 페이지 용량이 작다	인덱스페이지 용량이 크
30% 이내 사용 권장	3% 이내 사용 권장
1개	249개

[Spring] 트랜잭션 전파 옵션 REQUIRES_NEW를 사용해서 문제를 해결해보자

이번에는 REQUIRES_NEW를 사용해서 문제 상황을 가정하고 해결해보도록 하자. 문제 상황 회원가입에 성공하면 회원 DB와 회원 이력 로그 DB에 저장하는 비지니스 로직을 수행해야 한다. 회원가입과 이력 로그를 저장하는 로직을 하나의 트랜잭션으로 묶어서 처리하던 중, 만약 이력 로

https://hyuuny.tistory.com/188