11. 트랜잭션

#0.강의/2.데이터베이스로드맵/2.기본

- /트랜잭션이 필요한 이유
- /커밋, 롤백
- /트랜잭션의 ACID 속성
- /트랜잭션 격리 수준
- /정리

트랜잭션이 필요한 이유

우리는 제약 조건을 통해 데이터베이스의 '상태'가 항상 유효하도록 지키는 법을 배웠다. 가격이 음수가 되거나, 존재하지 않는 회원의 주문이 생기는 것을 막았다.

하지만 데이터의 일관성을 위협하는 것은 잘못된 '상태'뿐만이 아니다. 데이터를 변경하는 '행위'의 과정에서 문제가 생길 수도 있다.

우리 쇼핑몰에서 고객이 '주문하기' 버튼을 누르는, 가장 중요하고 흔한 시나리오를 생각해 보자. 이 하나의 비즈니스 행위 뒤에서, 데이터베이스는 최소한 두 가지의 중요한 작업을 순차적으로 처리해야 한다.

1. **작업 1**: `orders` 테이블에 새로운 주문 정보를 `INSERT` 한다.
2. **작업 2**: `products` 테이블에서 방금 주문된 상품의 재고(`stock_quantity`)를 `UPDATE` 하여 1만큼 줄인다. (정확히는 주문 수만큼 줄인다.)

이 두 작업은 논리적으로 **절대 쪼개질 수 없는 하나의 묶음**이다.

만약 트랜잭션이 없다면? (재앙의 시나리오)

만약 이 두 작업이 아무런 안전장치 없이 그냥 순서대로 실행된다고 가정해 보자.

시나리오: 1번 작업은 성공, 2번 작업은 실패

`INSERT` 쿼리로 `orders` 테이블에 새로운 주문이 성공적으로 기록되었다. 이제 `UPDATE` 쿼리로 `products` 테이블의 재고를 줄이려는 바로 그 찰나, 애플리케이션 서버에 문제가 발생하거나, 데이터베이스 서버의 전원이 나가거나, 네트워크 연결이 끊기거나, 디스크에 오류가 발생하는 등 예기치 못한 문제가 발생하여 2번 작업이 실패했다면 어떻게 될까?

데이터베이스는 다음과 같은 아주 위험한, **일관성이 깨진(inconsistent)** 상태에 빠지게 된다.

- `orders` **테이블**: 주문이 기록되어 있다. (회사는 돈을 받을 것이고, 고객은 물건을 받을 것이라 기대한다.)
- `products` **테이블**: 재고가 줄어들지 않았다. (회사는 여전히 팔 수 있는 재고가 있다고 착각한다.)

이것은 심각한 문제를 야기한다.

- **재고 관리의 실패**: 실제로는 팔린 상품을 계속 판매 가능한 재고로 인식하여, 결국 재고가 없는데도 주문을 받는 '초과 판매' 상황이 발생한다.
- **고객 신뢰도 하락**: 재고가 없어 주문을 취소당한 고객은 우리 쇼핑몰에 대한 신뢰를 잃게 된다.
- **잘못된 데이터 분석**: 모든 매출, 재고 관련 리포트가 실제와 달라지므로 믿을 수 없게 된다.

반대의 경우도 마찬가지다. 2번 재고 감소가 먼저 성공하고, 1번 주문 기록이 실패하면 '주문자는 없는데 재고만 사라진' 유령 재고가 발생한다.

해결책: 트랜잭션 (Transaction)

이러한 재앙을 막기 위해 데이터베이스는 **트랜잭션(Transaction)**이라는 아주 중요한 개념을 제공한다.

트랜잭션이란, **논리적으로 절대 쪼개질 수 없는 하나 이상의 데이터베이스 작업 묶음(Unit of Work)**을 의미한다.

트랜잭션은 우리에게 아주 강력한 약속을 제공한다. 바로 **'전부 아니면 전무(All or Nothing)'** 라는 원칙이다.

- 트랜잭션으로 묶인 작업들은 **모두 다 성공해야만** 그 결과를 실제 데이터베이스에 영구적으로 반영한다.
- 만약 작업 그룹 내에서 **단 하나의 작업이라도 실패**하면, 그전에 성공했던 모든 작업들을 **전부 없었던 일로 되돌려 버린다.**

우리의 주문 처리 시나리오를 트랜잭션으로 묶으면 이렇게 동작한다.

1. 주문 처리 트랜잭션을 시작한다.
2. `INSERT` 문으로 `orders` 테이블에 주문을 기록한다. (성공)
3. `UPDATE` 문으로 `products` 테이블의 재고를 줄인다. (**실패!**)
4. 하나의 작업이라도 실패했으므로, 트랜잭션은 실패로 간주된다.
5. 데이터베이스는 2번에서 성공했던 `INSERT` 작업까지 **자동으로 취소(원상 복구)** 시킨다.
6. 결과적으로, `orders` 테이블과 `products` 테이블은 이 트랜잭션이 시작되기 전의 완벽하게 일관된 상태로 남게 된다.

트랜잭션은 이 'All or Nothing' 원칙을 통해, 어떠한 외부 장애 상황에서도 데이터베이스의 일관성과 안정성을 보장하는 핵심적인 안전장치다.

그렇다면 이 '전부 아니면 전무'라는 마법을 실제로 어떻게 구현할 수 있을까? 개발자가 데이터베이스에게 "지금부터 트랜잭션을 시작할게", "모든 작업이 성공했으니 영구 저장해줘", "문제가 생겼으니 전부 되돌려줘" 라고 명령하는 방법은 무엇일까?

다음 시간에는 트랜잭션을 제어하는 핵심 명령어인 `COMMIT` 과 `ROLLBACK` 에 대해 배우겠다.

커밋, 롤백

지난 시간에 우리는 여러 개의 작업을 하나의 묶음으로 처리하여 데이터의 일관성을 지키는 트랜잭션의 필요성에 대해 배웠다. '전부 아니면 전무(All or Nothing)'라는 트랜잭션의 약속은 데이터베이스의 안정성을 보장하는 핵심 원리다.

그렇다면 개발자는 데이터베이스에게 어떻게 "여기서부터 여기까지가 하나의 묶음이야" 라고 알려줄 수 있을까? 그리고 그 작업 묶음이 성공했을 때와 실패했을 때, 각각 어떻게 '영구 저장'과 '원상 복구'를 명령할 수 있을까?

이번 시간에는 트랜잭션을 직접 제어하는 세 가지 핵심 명령어를 배우겠다.

- `START TRANSACTION` : "지금부터 트랜잭션을 시작해라." 라고 데이터베이스에 선언하는 신호탄. 이 명령어 이후에 실행되는 모든 쿼리는 하나의 트랜잭션으로 묶여 관리된다. (`BEGIN` 이라고도 쓴다.)
- `COMMIT` : "트랜잭션 내의 모든 작업이 성공적으로 완료되었으니, 지금까지의 변경 사항을 디스크에 영구적으로 저장(반영)해라." 라는 의미의 최종 승인 명령어다. `COMMIT` 이 실행된 후에는 변경 내용을 되돌릴 수 없다.
- `ROLLBACK` : "문제가 발생하여 작업을 중단해라. 이 트랜잭션 내에서 실행했던 모든 변경 사항을 전부 취소하고, 트랜잭션이 시작되기 직전의 상태로 완벽하게 되돌려라." 라는 의미의 작업 취소(원상 복구) 명령어다.

실습: 계좌이체 시나리오

가장 고전적이고 이해하기 쉬운 계좌이체 시나리오를 통해 이 명령어들의 작동 방식을 직접 확인해 보자. A 고객의 계좌에서 B 고객의 계좌로 10,000원을 이체하는 상황이다.

실습 준비: `accounts` 테이블 생성

> **SQL 소스 파일 참고**
> 강의 자료가 PDF 파일이라 복잡한 SQL 코드를 복사할 때 오류가 발생할 수 있다.
> 이 경우, 섹션 1. 강의 소개와 수업 자료 ➔ SQL 소스 파일을 다운로드해서 사용하자.

```sql
DROP TABLE IF EXISTS accounts;
CREATE TABLE accounts (
    id INT PRIMARY KEY,
    owner_name VARCHAR(255) NOT NULL,
    balance INT NOT NULL
);

-- 초기 데이터 입력
INSERT INTO accounts (id, owner_name, balance) VALUES
(1, 'A고객', 50000),
(2, 'B고객', 20000);
```

시나리오 1: 성공적인 계좌이체 (`COMMIT`)

1. 거래 전 잔고 확인

먼저, 거래 전 두 고객의 잔고를 확인한다.

```sql
SELECT * FROM accounts;
```

[실행 결과]

id	owner_name	balance
1	A고객	50000
2	B고객	20000

2. 트랜잭션 시작 및 계좌이체 실행

이제 트랜잭션을 시작하고, 두 개의 `UPDATE` 문을 차례로 실행한다.

```sql
-- 트랜잭션 시작
START TRANSACTION;

-- 1번 작업: A고객 계좌에서 10,000원 출금
UPDATE accounts SET balance = balance - 10000 WHERE id = 1;
```

```sql
-- 2번 작업: B고객 계좌에 10,000원 입금
UPDATE accounts SET balance = balance + 10000 WHERE id = 2;
```

3. 최종 확정 (`COMMIT`)

두 작업이 모두 성공적으로 실행되었음을 확인하고, `COMMIT` 명령어로 변경 사항을 데이터베이스에 영구적으로 반영한다.

```sql
COMMIT;
```

4. 거래 후 최종 잔고 확인

`COMMIT` 이후에 다시 잔고를 확인해 보자.

```sql
SELECT * FROM accounts;
```

[실행 결과]

id	owner_name	balance
1	A고객	40000
2	B고객	30000

두 계좌의 잔고가 성공적으로 변경되어 영구 저장된 것을 확인할 수 있다.

시나리오 2: 실수로 인한 작업 취소 (`ROLLBACK`)

이번에는 A고객이 B고객에게 5,000원을 추가로 이체하려다가, 실수를 깨닫고 작업을 취소하는 상황을 가정해 보자.

1. 트랜잭션 시작 및 출금 실행

```sql
-- 새로운 트랜잭션 시작
START TRANSACTION;

-- 1번 작업: A고객 계좌에서 5,000원 출금
```

```sql
UPDATE accounts SET balance = balance - 5000 WHERE id = 1;
```

이 시점에서 A고객의 잔고는 임시적으로 35,000원이 되었다. 하지만 입금 전, "아차! 송금할 사람이 B가 아니었다!" 라는 사실을 깨달았다. 또는 무언가 오류가 발생해서 더는 진행하면 안되는 상황이 되었다.

현재 작업 상황을 확인해보자.

```sql
SELECT * FROM accounts;
```

[실행 결과]

id	owner_name	balance
1	A고객	35000
2	B고객	30000

- `balance - 5000` 연산으로 A 고객은 임시로 35000원이 되었다.

2. 작업 취소 (`ROLLBACK`)

이때 `ROLLBACK` 을 실행하여, 이번 트랜잭션에서 수행한 모든 작업을 없었던 일로 되돌린다.

```sql
ROLLBACK;
```

3. 원상 복구된 잔고 확인

`ROLLBACK` 이후에 다시 잔고를 확인해 보자.

```sql
SELECT * FROM accounts;
```

[실행 결과]

id	owner_name	balance
1	A고객	40000
2	B고객	30000

A고객의 잔고가 `UPDATE` 이전 상태인 40,000원으로 완벽하게 돌아온 것을 볼 수 있다. `ROLLBACK`이 출금 작업을 깨끗이 취소시킨 것이다.

실무 팁: MySQL의 `autocommit`

MySQL은 기본적으로 `autocommit` 모드가 활성화(`ON` 또는 `1`)되어 있다. 이것은 우리가 실행하는 모든 SQL 문 하나하나를 각각의 트랜잭션으로 간주하여, 성공적으로 실행되는 즉시 **자동으로** `COMMIT` 해버린다는 의미다.

우리가 위에서 한 것처럼 여러 개의 문장을 하나의 트랜잭션으로 묶고 싶다면, 반드시 `START TRANSACTION` 명령어로 명시적으로 트랜잭션의 시작을 알려야 한다. `START TRANSACTION`을 실행하면, 그 세션에서는 `COMMIT`이나 `ROLLBACK`을 만날 때까지 `autocommit`이 잠시 비활성화된다.

> **세션 용어**
>
> 세션은 데이터베이스 클라이언트(예: MySQL Workbench, DBeaver, 또는 터미널의 mysql 클라이언트)가 데이터베이스 서버에 연결된 순간부터 연결을 종료할 때까지의 논리적인 연결 단위를 말한다. 각 세션은 독립적인 작업 공간을 가지며, 한 세션에서 시작된 트랜잭션은 해당 세션 내에서만 유효하다.

`COMMIT`과 `ROLLBACK`은 데이터의 일관성을 지키는 개발자의 가장 중요한 두 가지 도구다. 이들의 역할을 명확히 이해하고 사용하는 것이 신뢰성 있는 서비스를 만드는 첫걸음이다.

우리는 방금 'All or Nothing'이라는 트랜잭션의 핵심 속성을 직접 눈으로 확인했다. 하지만 이것은 트랜잭션을 신뢰할 수 있게 만드는 4가지 속성 중 하나일 뿐이다. 데이터베이스 세계에서는 이 4가지 속성을 묶어 **ACID**라는 약어로 부른다.

다음 시간에는 모든 데이터베이스 개발자가 반드시 알아야 할 **ACID 원칙(원자성, 일관성, 격리성, 지속성)**에 대해 알아보겠다.

트랜잭션의 ACID 속성

우리는 `COMMIT`과 `ROLLBACK`을 통해 트랜잭션이 'All or Nothing' 원칙으로 동작하는 것을 확인했다. 이 원칙 덕분에 우리는 데이터베이스 작업 중 예기치 못한 문제가 발생해도 데이터가 일관성을 잃지 않을 것이라고 믿을 수 있게 되

었다.

사실 이 'All or Nothing'은 트랜잭션이 보장하는 4가지 특성 중 하나일 뿐이다. 데이터베이스 시스템 이론에서는 트랜잭션이 신뢰성을 갖추도록 반드시 지켜야 할 4가지 속성을 묶어 **ACID**라는 약어로 부른다. ACID는 모든 관계형 데이터베이스 시스템의 근간을 이루는 매우 중요한 개념이다.

ACID는 다음 네 단어의 첫 글자를 딴 것이다.

- **A**tomicity (원자성)
- **C**onsistency (일관성)
- **I**solation (격리성)
- **D**urability (지속성)

이 네 가지 속성이 모두 보장될 때, 우리는 비로소 그 데이터베이스의 트랜잭션을 '신뢰할 수 있다'고 말한다. 각각의 속성이 무엇을 의미하는지 우리의 계좌이체 예시를 통해 알아보자.

A: Atomicity (원자성)

"트랜잭션은 하나의 원자(Atom)처럼 더 이상 쪼갤 수 없는 논리적 단위이며, 전부 성공하거나 전부 실패한다."

이것이 바로 우리가 배운 'All or Nothing' 원칙이다. 계좌이체라는 트랜잭션은 'A 계좌 출금'과 'B 계좌 입금'이라는 두 개의 작업으로 이루어지지만, 데이터베이스는 이 둘을 하나의 원자적 작업으로 취급한다.

- **모두 성공 (`COMMIT`)**: 출금과 입금이 모두 성공적으로 끝나야만, 이체라는 트랜잭션 전체가 성공으로 기록된다.
- **모두 실패 (`ROLLBACK`)**: 출금은 성공했지만 입금이 실패하면, 성공했던 출금 작업까지 모두 취소되어 트랜잭션 전체가 실패로 돌아간다.

원자성 덕분에, 돈이 A의 계좌에서 사라졌는데 B의 계좌에는 나타나지 않는 '돈이 증발하는' 상태는 절대 발생하지 않는다.

C: Consistency (일관성)

"트랜잭션이 성공적으로 완료되면, 데이터베이스는 항상 일관된(valid) 상태를 유지해야 한다."

일관성은 트랜잭션의 실행 결과가 데이터베이스에 설정된 모든 규칙(제약 조건 등)을 위반하지 않음을 보장하는 속성이다.

- **계좌이체 예시**: 트랜잭션 시작 전, A(5만원)와 B(2만원) 계좌의 총액은 7만원이었다. 성공적인 트랜잭션

(`COMMIT`) 후, A(4만원)와 B(3만원) 계좌의 총액은 여전히 7만원이다. 트랜잭션이 시스템의 전체 돈의 총량을 바꾸지 않는다는 '일관성'을 유지했다.

- **제약 조건 예시**: 만약 `accounts` 테이블에 `CHECK (balance >= 0)` 이라는 제약 조건이 있다면, 잔고를 음수로 만드는 어떠한 트랜잭션도 결국 실패하고 `ROLLBACK` 될 것이다. 이로써 데이터베이스는 '잔고는 0 이상이어야 한다'는 일관성을 항상 지킬 수 있다.

원자성이 '과정'의 완결성을 보장한다면, 일관성은 '결과'의 유효성을 보장하는 속성이다.

I: Isolation (격리성)

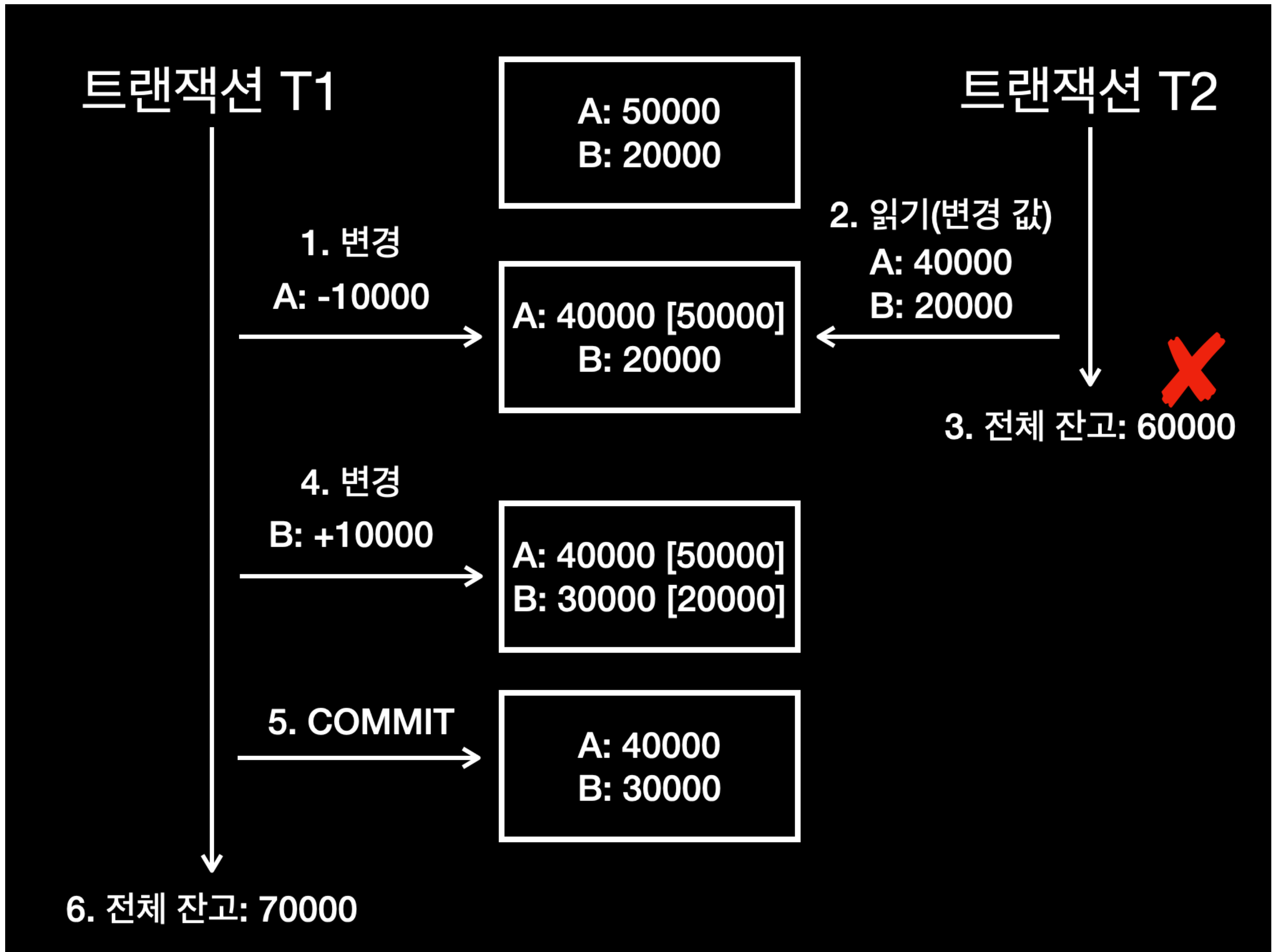
"하나의 트랜잭션이 실행 중일 때, 다른 트랜잭션이 해당 트랜잭션의 중간 결과에 끼어들어 간섭할 수 없다."

격리성은 여러 트랜잭션이 동시에 실행될 때, 마치 각각의 트랜잭션이 순서대로 하나씩 실행되는 것처럼 느끼게 해주는 특성이다.

- **상황**: A는 50000원 B는 20000원이 있다. A가 B에게 1만원을 이체하는 트랜잭션(T1)을 실행 중이다. `UPDATE` 로 A의 잔고는 4만원으로 바뀌었지만, 아직 `COMMIT` 은 하지 않아 B의 잔고는 2만원인 '중간 상태'다.
- **바로 그 순간**: 옆 창구에서 은행 직원이 전체 고객의 총잔고를 계산하는 트랜잭션(T2)을 실행한다.
- **격리성이 없다면?**: T2는 A의 잔고를 4만원으로, B의 잔고를 2만원으로 읽어서 총액이 6만원이라는 잘못된 결과를 얻게 된다. T1의 아직 완료되지 않은 불안정한 상태를 본 것이다. (이를 더티 리드(Dirty Read)라고 한다.)
- **격리성이 있다면?**: T2는 T1이 아직 `COMMIT` 되지 않았기 때문에, T1이 임시로 변경한 내용을 보지 못한다. T2는 T1이 시작되기 전의 값, 즉 A의 잔고 5만원과 B의 잔고 2만원을 읽어서 정확한 총액 7만원을 계산해낸다.

그럼 각각의 격리성이 있는 경우와 없는 경우를 각각 나누어 자세히 분석해보자.

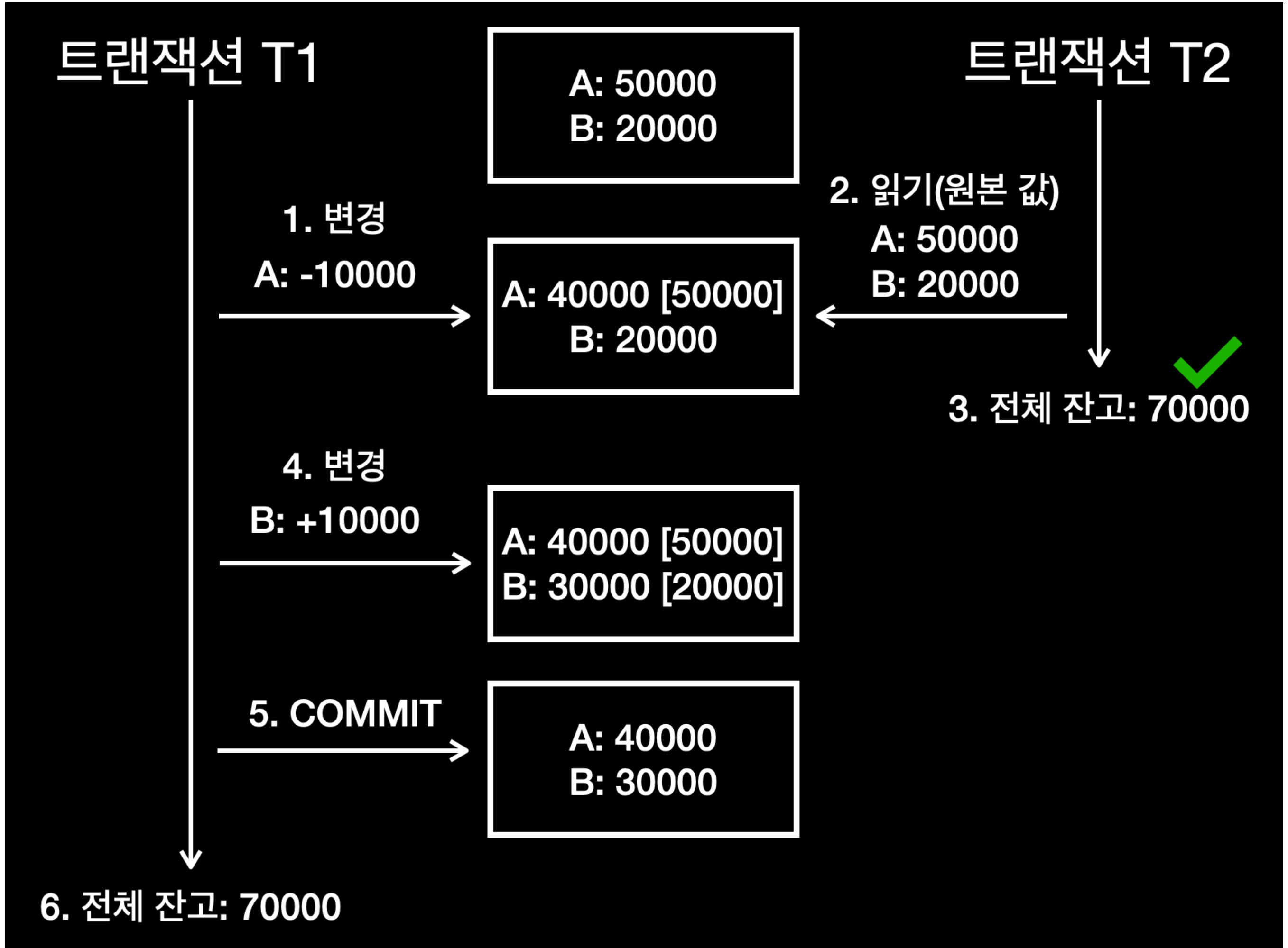
[격리성이 없는 경우]



1. A가 B에게 1만원을 이체하는 트랜잭션(T1)을 실행 중이다. `UPDATE` 로 A의 잔고는 4만원으로 바뀌었지만, 아직 `COMMIT` 은 하지 않아 B의 잔고는 2만원인 '중간 상태'다. 그림에서 오른쪽의 `[50000]` 의 값은 변경 전의 원본 값이다.
2. 바로 그 순간 옆 창구에서 은행 직원이 전체 고객의 총잔고를 계산하는 트랜잭션(T2)을 실행한다.
3. **격리성이 없다면?**: T2는 A의 잔고를 4만원으로, B의 잔고를 2만원으로 읽어서 총액이 6만원이라는 잘못된 결과를 얻게 된다. T1의 아직 완료되지 않은 불안정한 상태를 본 것이다. (이를 더티 리드(Dirty Read)라고 한다.)
4. T1은 B의 값에 10000원을 더한다. 그림에서 `[20000]` 의 값은 변경 전의 원본 값이다.
5. 트랜잭션을 커밋한다.
6. 전체 잔고를 다시 계산해보면 70000원이 된다.

여기서 문제가 발생한 이유는 T1이 원자적인 트랜잭션 작업을 하는 도중에 T2가 끼어들어서 T1의 값들을 볼 수 있기 때문이다.
결국 하나의 트랜잭션이 실행 중일 때, 다른 트랜잭션이 해당 트랜잭션의 중간 결과를 볼 수 있기 때문이다.
이 문제를 해결하려면 T1의 트랜잭션 작업을 다른 트랜잭션인 T2가 볼 수 없게 격리해야 한다.

[격리성이 있는 경우]



1. A가 B에게 1만원을 이체하는 트랜잭션(T1)을 실행 중이다. `UPDATE` 로 A의 잔고는 4만원으로 바뀌었지만, 아직 `COMMIT` 은 하지 않아 B의 잔고는 2만원인 '중간 상태'다. 그림에서 `[50000]` 의 값은 변경 전의 원본 값이다.
2. 바로 그 순간 옆 창구에서 은행 직원이 전체 고객의 총잔고를 계산하는 트랜잭션(T2)을 실행한다.
3. **격리성이 있다면?**: T2는 T1이 아직 `COMMIT` 되지 않았기 때문에, T1이 임시로 변경한 내용을 보지 못한다. T2는 T1이 시작되기 전의 값, 즉 A의 잔고 5만원과 B의 잔고 2만원을 읽어서 정확한 총액 7만원을 계산해낸다.
4. T1은 B의 값에 10000원을 더한다. 그림에서 `[20000]` 의 값은 변경 전의 원본 값이다.
5. 트랜잭션을 커밋한다.
6. 전체 잔고를 다시 계산해보면 70000원이 된다.

여기서는 T1이 원자적인 트랜잭션 작업을 하는 도중에 T2가 끼어들어서 T1의 값들을 볼 수 없다.
결국 하나의 트랜잭션이 실행 중일 때, 다른 트랜잭션이 해당 트랜잭션의 중간 결과를 볼 수 없는 것이다.
각각의 트랜잭션이 작업 중일 때 서로의 값을 볼 수 없게 격리한 덕분에 정상적인 값을 계산해 낼 수 있었다.

이처럼 격리성은 여러 사용자가 동시에 데이터베이스에 접근하는 환경에서 데이터가 엉망이 되는 것을 막아주는 매우 중요한 속성이다.

D: Durability (지속성)

"성공적으로 완료되어 `COMMIT` 된 트랜잭션의 결과는, 시스템에 장애가 발생하더라도 영구적으로 보존된다."

지속성은 `COMMIT` 된 데이터는 절대 사라지지 않는다는 약속이다.

- **상황**: A가 B에게 성공적으로 1만원을 이체하고, 은행 앱에서 "이체 완료" 메시지를 확인했다. (`COMMIT` 이 성공했다.)
- **1초 뒤**: 은행의 전체 전산 시스템이 다운되었다.
- **지속성 보장**: 시스템이 재부팅된 후에도, A의 잔고는 4만원, B의 잔고는 3만원인 상태가 그대로 유지되어 있어야 한다. `COMMIT` 된 결과는 데이터베이스의 저장소(SSD, HDD)에 있는 트랜잭션 로그 등에 기록되어, 어떤 장애에도 살아남을 수 있다.

이 네 가지 ACID 속성이 있기에, 우리는 안심하고 은행 거래를 하고, 쇼핑몰에서 주문을 하고, 데이터베이스에 우리의 소중한 데이터를 맡길 수 있는 것이다.

이 중, 특히 '격리성'은 "어느 정도로 엄격하게 격리(고립)시킬 것인가?"에 따라 몇 가지 단계, 즉 **격리 수준(Isolation Level)**으로 나뉜다. 완벽하게 격리시키면 데이터는 매우 안전하지만 동시에 많은 작업을 처리하기 어려워져 성능이 저하되고, 격리 수준을 낮추면 성능은 올라가지만 특정 데이터 불일치 문제가 발생할 수 있다.

다음 시간에는 이 중요한 트레이드오프 관계인 **트랜잭션 격리 수준**에 대해 간단히 소개하겠다.

트랜잭션 격리 수준

지난 시간에 우리는 ACID 원칙 중 하나인 격리성(Isolation)에 대해 배웠다. 격리성은 여러 트랜잭션이 동시에 실행될 때 서로 간섭하지 못하게 막아주는 중요한 속성이다.

여기서 실무적인 질문이 하나 나온다. "완벽한 격리는 항상 좋은 것일까?"
만약 모든 트랜잭션이 완벽하게 격리(고립)되어, 마치 한 줄로 줄을 서서 한 번에 하나씩만 순서대로 실행된다면 데이터는 매우 안전할 것이다. 하지만 수천, 수만 명의 사용자가 동시에 접속하여 주문하고, 상품을 조회하는 온라인 쇼핑몰은 어떻게 될까? 아마 거북이처럼 느려져서 아무도 사용하지 않으려 할 것이다.

이것이 바로 데이터베이스 개발자가 항상 마주하는 중요한 트레이드오프다.
데이터 정합성(Correctness) vs 동시성/성능(Concurrency/Performance)

데이터베이스는 이 트레이드오프를 개발자가 직접 '조절'할 수 있도록 **트랜잭션 격리 수준(Transaction Isolation**

Level)이라는 몇 가지 단계를 제공한다. 격리 수준을 높이면(엄격하게 만들면) 데이터 정합성은 올라가지만 동시성이 떨어져 성능이 저하되고, 격리 수준을 낮추면(느슨하게 만들면) 성능은 올라가지만 특정 데이터 부정합 문제가 발생할 수 있다.

> 🗒 **트랜잭션 격리 수준은 지금 단계에서는 너무 깊이있게 들어가기보다는 대략 이런 개념이 있다는 정도만 알아두면 충분하다. 더 자세한 내용은 이후 데이터베이스 성능 최적화와 같은 심화 과정에서 상세히 다룰 예정이다.**

동시성 문제 (Concurrency Problems)

격리 수준이 낮을 때 발생할 수 있는 대표적인 문제 현상(Anomaly)들은 다음과 같다. 개념만 가볍게 이해해 보자.

1. **더티 리드 (Dirty Read)**
   - **정의**: 한 트랜잭션이 아직 `COMMIT` 하지 않은, 수정 중인 데이터를 다른 트랜잭션이 읽는 것.
   - **예시**: 트랜잭션 A가 특정 상품의 가격을 100원에서 120원으로 바꾸고 아직 `COMMIT` 하지 않았다. 이때 트랜잭션 B가 이 상품의 가격을 조회했더니 '120원'이 보였다. 하지만 잠시 후 트랜잭션 A가 작업을 취소(`ROLLBACK`)해버리면, 가격은 다시 100원이 된다. 트랜잭션 B는 결국 존재하지도 않는 '더러운' 데이터를 읽은 셈이다.

2. **반복 불가능 읽기 (Non-Repeatable Read)**
   - **정의**: 한 트랜잭션 내에서 똑같은 `SELECT` 쿼리를 두 번 실행했는데, 그 사이에 다른 트랜잭션이 값을 수정하고 `COMMIT`하는 바람에 두 쿼리의 결과가 다르게 나오는 현상.
   - **예시**: 트랜잭션 A가 특정 상품의 재고 필드 값이 '10개'인 것을 확인했다. 잠시 다른 작업을 하다가 다시 재고를 확인했더니, 그 사이에 다른 트랜잭션 B가 그 상품을 하나 사가서 재고 필드 값이 '9개'로 바뀌어 있다. 트랜잭션 A 안에서 같은 데이터의 반복 조회가 불가능해진 것이다.

3. **유령 읽기 (Phantom Read)**
   - **정의**: 한 트랜잭션 내에서 특정 범위의 데이터를 두 번 읽었는데, 첫 번째 조회에서는 없었던 새로운 행(유령)이 두 번째 조회에서 나타나는 현상. 다른 트랜잭션이 새로운 행을 `INSERT`하고 `COMMIT`했기 때문에 발생한다.
   - **예시**: 트랜잭션 A가 '전자기기' 카테고리의 상품 수를 세었더니 '5개'였다. 잠시 후 똑같이 수를 세었더니, 그 사이에 다른 트랜잭션 B가 새로운 '전자기기' 상품을 등록해서 '6개'가 되었다. 없었던 유령 상품이 나타난 것이다.

4가지 표준 격리 수준

SQL 표준은 이러한 문제들을 방지하는 강도에 따라 4가지 격리 수준(Isolation Level)을 정의한다. 아래로 갈수록 격리 수준이 높아지고, 성능은 떨어진다.

격리 수준	Dirty Read	Non-Repeatable Read	Phantom Read
`READ UNCOMMITTED`	발생	발생	발생
`READ COMMITTED`	방지	발생	발생
`REPEATABLE READ`	방지	방지	발생(일부 방지*)
`SERIALIZABLE`	방지	방지	방지

- `READ UNCOMMITTED`: 거의 아무것도 막아주지 않는, 가장 낮은 수준. 정합성 이슈가 많아 거의 사용되지 않는다.
- `READ COMMITTED`: 더티 리드를 방지한다. 즉, `COMMIT` 된 데이터만 읽을 수 있다. 오라클, SQL Server 등 많은 데이터베이스의 기본 격리 수준이다.
- `REPEATABLE READ`: 한 트랜잭션 안에서는 데이터의 일관된 조회를 보장해 준다. **MySQL의 InnoDB 스토리지 엔진이 사용하는 기본 격리 수준**이다.
- `SERIALIZABLE`: 가장 엄격한 수준. 동시성 문제를 완벽하게 차단하지만, 트랜잭션을 거의 순서대로 실행시켜 동시 처리 성능이 가장 낮다.

> **최신 MySQL은 InnoDB 스토리지 엔진을 기본으로 사용한다.**
> 별도의 이야기가 없으면 InnoDB 스토리지 엔진이라고 생각하면 된다.

> **심화 - MySQL InnoDB 특징**
> SQL 표준에 따르면 REPEATABLE READ 수준에서는 팬텀 리드(Phantom Read)가 발생할 수 있다고 정의된다.
> *MySQL의 InnoDB 엔진은 **MVCC와 갭 락(Gap Lock)이라는 기술을 통해** REPEATABLE READ **수준에서도 대부분의 팬텀 리드가 발생하는 것을 막아준다.** 이 때문에 InnoDB의 REPEATABLE READ는 표준의 SERIALIZABLE에 가까운 격리성을 제공한다. (가깝다는 것이지 완벽한 것은 아니다.)

격리 수준 확인 및 변경하기

현재 격리 수준을 확인하고, 필요에 따라 변경하는 SQL을 알아보자.

현재 격리 수준 확인

현재 세션(연결)의 격리 수준을 확인하는 쿼리다.

```
SELECT @@transaction_isolation;
```

[실행 결과]

@@transaction_isolation
REPEATABLE-READ

격리 수준 변경

격리 수준은 특정 트랜잭션에만 적용하거나, 현재 세션 내내 적용하거나, 데이터베이스 시스템 전체에 적용할 수 있다.

현재 세션에서만 변경하기 (가장 일반적)

`SET SESSION` 옵션을 사용하면, 지금 내가 접속한 이 세션에만 격리 수준이 변경된다. 다른 개발자나 사용자에게는 영향을 주지 않으므로 비교적 안전하다.

```
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

```
SELECT @@transaction_isolation;
```

[실행 결과]

@@transaction_isolation
READ-COMMITTED

- READ-COMMITTED으로 변경되었다.

글로벌(전역)으로 변경하기 (주의!)

`SET GLOBAL`은 데이터베이스 서버의 기본 격리 수준 자체를 변경한다. 이후에 연결되는 모든 세션에 영향을 준다.

```
SET GLOBAL TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

`SET GLOBAL` 명령어는 서버 재시작 시 사라지는 임시적인 설정이다. 이 설정을 영구적으로 설정하려면 MySQL 설정 파일 (`my.cnf` 또는 `my.ini`)을 변경해야 한다.

시스템 전체에 영향을 주는 변경은 매우 신중해야 하므로, 특별한 정책이 있지 않다면 기본값을 따르는 것이 좋다.

실무 팁: 어떤 격리 수준을 선택해야 할까?

이론을 배웠으니 이제 가장 중요한 실무적인 질문에 답을 할 차례다. "그래서 우리 쇼핑몰 프로젝트에는 어떤 격리 수준을 써야 할까?"

결론부터 말하자면, **특별한 이유가 없다면 MySQL의 기본 격리 수준인** `REPEATABLE READ`**를 그대로 사용하는 것을 권장한다.** 트래픽이 매우 많고 단순 조회 위주의 웹 애플리케이션이라면 `READ COMMITTED`을 고려할 수 있다.

데이터베이스를 만드는 회사들이 기본값을 정할 때는 다 그만한 이유가 있다. `REPEATABLE READ`는 데이터 정합성과 동시성 사이에서 매우 합리적인 균형을 제공한다. 한 트랜잭션이 시작되면 그 트랜잭션이 끝날 때까지 다른 트랜잭션의 변경 사항에 영향을 받지 않고 일관된 데이터를 조회할 수 있다. 심지어 MySQL의 InnoDB 엔진은 갭 락(Gap Lock) 덕분에 다른 데이터베이스라면 막지 못했을 팬텀 리드까지 대부분 막아주니, 웬만한 비즈니스 로직에서는 충분히 안정적이다.

그렇다면 언제 다른 격리 수준을 **'고민'** 해볼 수 있을까?

`READ UNCOMMITTED`로 낮추는 것을 고려하는 경우

이 격리 수준은 실무에서 **거의 사용하지 않는다**. 왜냐하면 커밋되지 않은, 즉 언제든 사라질 수 있는 '더러운' 데이터(Dirty Data)를 읽는 것을 허용하기 때문이다. 이는 데이터 정합성에 매우 심각한 문제를 일으킬 수 있다.
그리고 MySQL을 포함한 현대의 데이터베이스들은 다양한 최적화 기법을 제공하기 때문에 다음 단계인 `READ COMMITTED`와 성능 차이가 거의 나지 않는다. (약 0 ~ 3%)

물론 이론적이긴 하지만, 아주 예외적인 상황에서, 성능 향상을 위해 '정확성'을 어느 정도 희생할 수 있을 때만 제한적으로 고려해 볼 수 있다.

어떤 경우일까? 바로 **대용량 데이터에 대한 실시간 집계나 통계 작업**을 수행할 때이다.

쇼핑몰의 `items` 테이블에 수억 개의 상품이 있고, 마케팅팀에서 '전자기기' 카테고리 상품의 **대략적인** 평균 가격 추이를 실시간으로 보고 싶어 한다고 상상해보자. 이 경우, 몇몇 상품의 가격이 트랜잭션 도중 잠깐 바뀌었다가 롤백되더라도 전체 평균 가격에 미치는 영향은 아주 미미할 것이다. 마케팅팀에 필요한 것은 회계 장부처럼 1원 단위까지 정확한 수치가 아니라, 현재 시장의 '경향성'을 파악하는 것이기 때문이다.

이럴 때 `READ UNCOMMITTED`를 사용하면, 다른 트랜잭션이 특정 상품의 가격을 수정하는 동안 락(Lock)을 기다리지 않고 즉시 데이터를 읽어올 수 있다.

예를 들어보자.

1. **트랜잭션 1 (운영팀)**: '게이밍 노트북' 가격을 1,500,000원에서 1,450,000원으로 변경하는 작업을 시작한다. 아직 **커밋(COMMIT)은 하지 않았다.**

   ```
   -- 트랜잭션 시작
   START TRANSACTION;
   UPDATE items SET price = 1450000 WHERE item_name = '게이밍 노트북';
   ```

2. **트랜잭션 2 (마케팅팀 분석 쿼리)**: 바로 그 시점에, `READ UNCOMMITTED` 격리 수준으로 설정된 트랜잭션에서 '전자기기'의 평균 가격을 조회한다.

   ```
   SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
   SELECT AVG(price) FROM items WHERE category = '전자기기';
   ```

   이 쿼리는 트랜잭션 1이 아직 커밋하지 않은 1,450,000원이라는 가격을 그대로 읽어서 평균을 계산한다.

3. **트랜잭션 1 (운영팀)**: 운영팀 직원이 실수를 깨닫고 작업을 취소(**ROLLBACK**)한다. '게이밍 노트북' 가격은 원래대로 1,500,000원으로 돌아간다.

   ```
   ROLLBACK;
   ```

결과적으로 마케팅팀이 본 평균 가격은 '세상에 잠시 존재했다가 사라진' 유령 데이터를 기반으로 계산된 것이다. 하지만 수만 개의 '전자기기' 상품 중 단 하나의 데이터가 잠시 바뀐 것이 전체 평균에 큰 영향을 주지는 않으며, 빠른 응답 속도

가 더 중요한 가치일 수 있다.

하지만 다시 한번 강조한다. 이는 매우 위험한 선택이다. 주문 처리, 재고 관리, 고객 정보 조회와 같이 데이터의 정확성이 생명인 핵심 비즈니스 로직에서는 절대 사용해서는 안 된다. `Dirty Read`로 인해 고객에게 잘못된 가격을 보여주거나 재고가 있는데도 없다고 판단하는 등의 끔찍한 상황이 발생할 수 있다.

결론적으로 `READ UNCOMMITTED`는 '이런 것도 있구나' 정도로만 알아두고, 실무에서는 사용을 지양하는 것이 현명하다. 통계나 분석 작업이 필요하다면, 운영 데이터베이스(Source a.k.a Primary)를 복제한 분석용 데이터베이스(Replica)를 따로 구축해서 그곳에서 자유롭게 쿼리를 실행하는 것이 훨씬 더 안전하고 일반적인 방법이다.

`READ COMMITTED`로 낮추는 것을 고려하는 경우
성능과 동시성 확보가 매우 중요하고, 약간의 데이터 비일관성을 감수할 수 있는 특정 상황에서 `READ COMMITTED`를 선택할 수 있다.

많은 온라인 서비스, 특히 웹 애플리케이션은 '하나의 요청 = 하나의 짧은 트랜잭션'으로 동작하는 경우가 대부분이다. 예를 들어, 사용자가 상품 목록을 조회하면, 서버는 DB에서 상품 목록을 조회해서 반환하고 트랜잭션을 바로 종료한다. 이런 환경에서는 트랜잭션 하나에서 똑같은 `SELECT` 쿼리를 여러 번 실행할 일이 거의 없으므로, `Non-Repeatable Read`가 발생할 가능성 자체가 낮다.

이럴 때 `READ COMMITTED`를 사용하면 불필요한 락(Lock)들을 사용하지 않게 되어 락(Lock)을 기다리는 시간이 줄어든다. 결과적으로 데이터베이스의 전체적인 처리량(Throughput)이 향상될 수 있다. 오라클(Oracle), PostgreSQL, SQL Server와 같은 다른 인기 있는 데이터베이스들이 `READ COMMITTED`를 기본값으로 채택한 이유이기도 하다.

성능 향상은 애플리케이션의 성격에 따라 다르지만, 업데이트와 입력이 빈번하게 발생하는 높은 동시성 환경에서는 대략 10% ~ 30% 수준의 처리량 향상을 기대해 볼 수 있으며, 경우에 따라 그 이상도 가능하다. 물론, 이는 일반적인 수치이며 실제로는 애플리케이션의 쿼리 패턴과 데이터 경쟁 정도에 따라 크게 달라질 수 있다.

`SERIALIZABLE`로 높이는 것을 고려하는 경우
데이터의 정합성이 극도로 중요해서 아주 작은 예외도 허용할 수 없을 때 사용한다. 이는 마치 트랜잭션을 한 줄로 세워 순서대로 처리하는 것과 유사하므로, 동시성이 크게 저하된다.

예를 들어, 여러 사용자가 동시에 하나의 은행 계좌에서 돈을 인출하려 하거나, 초당 수십 건의 주문이 몰리는 한정 수량 상품의 재고를 차감하는 로직이 있다면 `SERIALIZABLE`을 고려해볼 수 있다.(실무에서는 다른 대안을 선택한다.)

하지만 그 대가는 매우 혹독하다. 동시성이 현저히 떨어져서 시스템 전체 성능에 병목이 될 가능성이 매우 높다. 따라서 `SERIALIZABLE` 은 정말 다른 방법이 없을 때 사용하는 최후의 수단으로 생각해야 한다.

대부분의 경우, 격리 수준을 무작정 높이기보다는 애플리케이션 코드 레벨에서 `SELECT ... FOR UPDATE` 와 같은 비관적 락(Pessimistic Lock)을 사용하거나, 버전 번호를 두는 낙관적 락(Optimistic Lock)을 구현하여 문제를 더 정교하게 해결하는 것이 현명한 선택이다.

이런 이유로 `SERIALIZABLE` 은 실무에서 거의 사용하지 않는다.

이러한 격리 수준은 데이터베이스의 동시성 제어(Concurrency Control)와 관련된 매우 깊이 있고 중요한 주제다. 각 수준의 동작 원리와 그에 따른 성능 특성을 이해하는 것은 고급 데이터베이스 개발자와 관리자에게 중요하다.

> **참고**
> 트랜잭션 격리 수준과 락에 대한 내용은 지금 단계에서는 너무 깊이있게 들어가기보다는 대략 이런 개념이 있다는 정도만 알아두면 충분하다. 더 자세한 내용과 추가로 데이터베이스 락에 대한 내용은 이후 데이터베이스 성능 최적화와 같은 심화 과정에서 상세히 다룰 예정이다.

정리

트랜잭션이 필요한 이유

- 데이터 변경 시 여러 작업을 논리적으로 쪼갤 수 없는 하나의 묶음(Unit of Work)으로 다루기 위해 트랜잭션이 필요하다.
- 주문 생성(INSERT)과 재고 감소(UPDATE)처럼 두 작업 중 하나만 성공하면 데이터 일관성이 깨지는 심각한 문제가 발생한다.
- 트랜잭션은 '전부 아니면 전무(All or Nothing)' 원칙을 보장한다.
- 묶인 작업 중 하나라도 실패하면, 이전에 성공한 모든 작업을 자동으로 취소(원상 복구)하여 데이터의 일관성과 안정성을 지킨다.

커밋, 롤백

- 트랜잭션은 `START TRANSACTION`, `COMMIT`, `ROLLBACK` 명령어로 제어한다.
- `START TRANSACTION`: 트랜잭션의 시작을 데이터베이스에 알린다.

- `COMMIT`: 트랜잭션 내의 모든 작업이 성공했을 때, 변경 사항을 영구적으로 저장한다.
- `ROLLBACK`: 문제가 발생했을 때, 트랜잭션 내에서 실행한 모든 변경 사항을 취소하고 시작 전 상태로 되돌린다.
- MySQL은 기본적으로 `autocommit` 모드가 활성화되어 있어, 모든 SQL 문이 즉시 커밋되므로 여러 문장을 묶으려면 `START TRANSACTION`을 명시적으로 사용해야 한다.

트랜잭션의 ACID 속성

- 신뢰할 수 있는 트랜잭션은 원자성(Atomicity), 일관성(Consistency), 격리성(Isolation), 지속성(Durability)이라는 ACID 속성을 반드시 만족해야 한다.
- **원자성 (Atomicity)**: 트랜잭션은 전부 성공하거나 전부 실패해야 한다. (All or Nothing)
- **일관성 (Consistency)**: 트랜잭션 완료 후에도 데이터베이스는 제약 조건 등 유효한 상태를 유지해야 한다.
- **격리성 (Isolation)**: 여러 트랜잭션이 동시에 실행될 때, 서로의 작업 중간 결과에 간섭할 수 없다.
- **지속성 (Durability)**: 성공적으로 `COMMIT`된 트랜잭션의 결과는 시스템 장애가 발생해도 영구적으로 보존된다.

트랜잭션 격리 수준

- 격리 수준은 데이터 정합성과 동시성(성능) 사이의 트레이드오프를 조절하는 설정이다.
- 격리 수준이 낮으면 동시성 문제는 발생할 수 있으나 성능이 향상되고, 높이면 정합성은 보장되나 성능이 저하된다.
- **동시성 문제 유형**: 더티 리드(Dirty Read), 반복 불가능 읽기(Non-Repeatable Read), 유령 읽기(Phantom Read) 등이 있다.
- **표준 격리 수준**: `READ UNCOMMITTED`, `READ COMMITTED`, `REPEATABLE READ`, `SERIALIZABLE` 순으로 격리 강도가 높아진다.
- **MySQL InnoDB 엔진**의 기본 격리 수준은 `REPEATABLE READ`이며, 대부분의 비즈니스 환경에서 정합성과 성능의 합리적인 균형을 제공하므로 특별한 이유가 없다면 기본값을 사용하면 된다. 트래픽이 매우 많고 반복 불가능 읽기, 유령 읽기 상황이 자주 발생하지 않는 웹 애플리케이션이라면 `READ COMMITTED`을 고려할 수 있다.