애플리케이션에서 잠금 사용해 보기

2025.09.03.

System Software Lab.

목차

01 잠금 전략 없이 로직 실행해 보기

02 Java 동기화 키워드 활용

03 데이터베이스 잠금 전략 활용

- a 비관적 잠금
- b 낙관적 잠금

01

잠금 전략 없이 실행해 보기

만들어 볼 것

// TODO: 사용자는 상품을 구매할 수 있다.

→ 사용자가 결제한다. (결제 API 호출)

→ 상품 재고를 구매 수량만큼 감소시킨다.

→ 사용자의 주문한 목록에 추가한다.

등등...

데이터베이스 테이블 설계

id	product_id	quantity
주요 키 *(레코드를 유일하게 식별할 수 있는 값)*	상품 외래 키 *(다른 테이블과 연결을 위한 다른 테이블의 주요 키)*	상품 수량

엔티티 데이터베이스 테이블에 저장된 레코드*(튜플)*과 연결되는 Java 객체

```java
@Entity
public class Stock {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private Long productId;

    private Long quantity;
}
```

비즈니스 로직

// TODO: 사용자는 상품을 구매할 수 있다.

→ 상품 재고를 구매 수량만큼 감소시킨다.

엔티티 데이터베이스 테이블에 저장된 레코드*(튜플)*과 연결되는 Java 객체

```java
@Entity
public class Stock {
    // id, productId, quantity

    public void decrease(Long quantity) {
        if (this.quantity - quantity < 0) {
            throw new RuntimeException("재고는 0개 미만이 될 수 없습니다.");
        }

        this.quantity -= quantity;
    }
}
```

비즈니스 로직

리포지토리 엔티티 클래스를 참고하여 데이터의 CRUD를 처리하는 인터페이스로, 그 구현체는 Spring이 자동으로 생성함

```
public interface StockRepository extends JpaRepository<Stock, Long> {

}
```

비즈니스 로직

서비스 사용자의 요청이 오면 비즈니스 로직을 처리하는 Java 클래스 및 Spring 빈 컨테이너가 생성된 인스턴스가 참조

```java
@Service
public class StockService {
    private final StockRepository stockRepository;

    public StockService(StockRepository stockRepository) {
        this.stockRepository = stockRepository;
    }

    @Transactional
    public void decrease(Long id, Long quantity) {
        // Stock 조회
        Stock stock = stockRepository.findById(id)
                .orElseThrow(() -> new RuntimeException("재고 정보가 존재하지 않습니다."));

        // 재고를 감소시킨 후
        stock.decrease(quantity);

        // 갱신된 값을 저장
        stockRepository.saveAndFlush(stock);
    }
}
```

비즈니스 로직 테스트

테스트 비즈니스 로직을 코드로 평가할 수 있는 Java 클래스

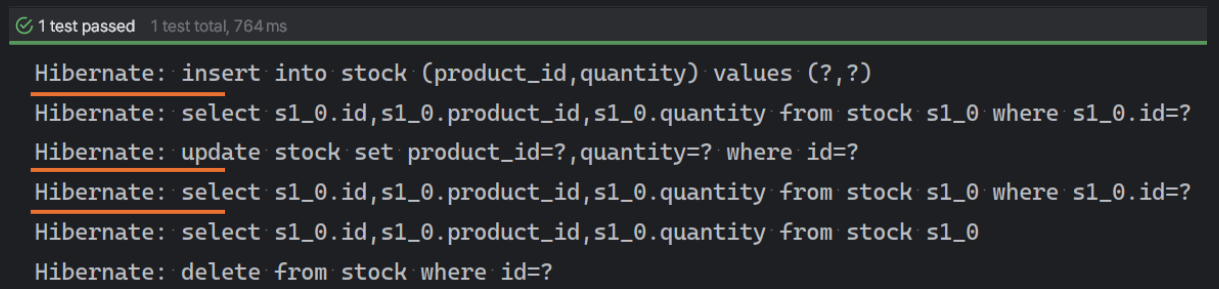
```java
@BeforeEach
void beforeEach() {
    stockRepository.saveAndFlush(new Stock(1L, 100L));
}

@AfterEach
void afterEach() {
    stockRepository.deleteAll();
}

@Test
void 재고감소() {
    stockService.decrease(1L, 1L);

    Stock stock = stockRepository.findById(1L).orElseThrow();

    assertThat(stock.getQuantity()).isEqualTo(99L);
}
```



비즈니스 로직 실행 시 발생할 수 있는 문제

테스트 비즈니스 로직을 코드로 평가할 수 있는 Java 클래스

```java
@Test
void 단시간에_100개의_요청() throws InterruptedException {
    int threadCount = 100;
    ExecutorService executorService = Executors.newFixedThreadPool(32);
    CountDownLatch latch = new CountDownLatch(threadCount);

    for (int i = 0; i < threadCount; ++i) {
        executorService.submit(() -> {
            try {
                stockService.decrease(1L, 1L);
            } finally {
                latch.countDown();
            }
        });
    }

    latch.await();

    Stock stock = stockRepository.findById(1L).orElseThrow();

    assertThat(stock.getQuantity()).isEqualTo(0L);
}
```

```
1 test failed  1 test total, 1 sec 52 ms
org.opentest4j.AssertionFailedError:
expected: 0L
 but was: 95L
Expected :0L
Actual   :95L
<Click to see difference>
```

발생하는 문제의 원인

정상 실행 시

스레드 1	Stock	스레드 2
`select * from stock where id = 1`	`{id: 1, quantity: 5}`	
`update set quantity = 4 from stack where id = 1`	`{id: 1, quantity: 4}`	
	`{id: 1, quantity: 4}`	`select * from stock where id = 1`
	`{id: 1, quantity: 3}`	`update set quantity = 3 from stack where id = 1`

스레드 간 경쟁 발생 시

스레드 1	Stock	스레드 2
`select * from stock where id = 1`	`{id: 1, quantity: 5}`	
	`{id: 1, quantity: 5}`	`select * from stock where id = 1`
`update set quantity = 4 from stack where id = 1`	`{id: 1, quantity: 4}`	
	`{id: 1, quantity: 4}`	`update set quantity = 4 from stack where id = 1`

02

Java 동기화 키워드 활용

Java synchronized 키워드 활용

서비스 사용자의 요청이 오면 비즈니스 로직을 처리하는 Java 클래스 및 Spring 빈 컨테이너가 생성된 인스턴스가 참조

```java
@Transactional
public synchronized void decrease(Long id, Long quantity) {
    // Stock 조회
    Stock stock = stockRepository.findById(id)
            .orElseThrow(() -> new RuntimeException("재고 정보가 존재하지 않습니다."));

    // 재고를 감소시킨 후
    stock.decrease(quantity);

    // 갱신된 값을 저장
    stockRepository.saveAndFlush(stock);
}
```

```
1 test failed  1 test total, 1 sec 195 ms

org.opentest4j.AssertionFailedError:
expected: 0L
 but was: 50L
Expected :0L
Actual   :50L
<Click to see difference>
```

Transactional 애너테이션 메서드에서 발생하는 쿼리를 하나의 트랜잭션으로 묶어주는 기능

```java
public void decrease(Long id, Long quantity) {
    BEGIN();
    stockService.decrease(id, quantity);
    COMMIT();
}
```

다른 스레드에서 COMMIT();이 실행돼 예상 값과 다른 결과가 저장될 수 있음

decrease 메서드에만 synchronized 키워드가 적용

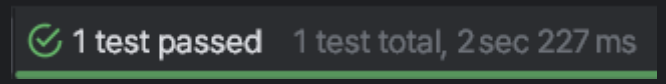
Java synchronized 키워드 활용

서비스 사용자의 요청이 오면 비즈니스 로직을 처리하는 Java 클래스 및 Spring 빈 컨테이너가 생성된 인스턴스가 참조

```java
public synchronized void decrease(Long id, Long quantity) {
    // Stock 조회
    Stock stock = stockRepository.findById(id)
            .orElseThrow(() -> new RuntimeException("재고 정보가 존재하지 않습니다."));

    // 재고를 감소시킨 후
    stock.decrease(quantity);

    // 갱신된 값을 저장
    stockRepository.saveAndFlush(stock);
}
```



Transactional 애너테이션을 제거하고 synchronized 키워드를 사용할 경우 문제

- 메서드 내 쿼리가 트랜잭션으로 묶이지 않아 예외가 발생하면 메서드의 이전 로직을 되돌릴 수 없음
- 트래픽 분산 등의 이유로 여러 대의 애플리케이션(프로세스)을 실행하면 여전히 경쟁이 발생할 수 있음

03

비관적 잠금

비관적 잠금 *(Pessimistic Lock)*

스레드 1	Stock	스레드 2
`select * from stock where id = 1`	`{id: 1, quantity: 5}`	획득 시도하나 스레드 1에서 점유 중이므로 대기함
`update set quantity = 4 from stack where id = 1`	`{id: 1, quantity: 4}`	
	`{id: 1, quantity: 4}`	`select * from stock where id = 1`
	`{id: 1, quantity: 3}`	`update set quantity = 3 from stack where id = 1`

- 레코드에 배제적 잠금*(쓰기 잠금)*을 설정하여 **데이터 정합성을 유지**할 수 있음
- 한 트랜잭션에서 잠금을 설정하면, 다른 트랜잭션에서 잠금 해제 전까지 레코드를 갱신할 수 없음

- A 트랜잭션에서 a에 잠금을 설정하고, B 트랜잭션에서 b에 잠금을 설정했을 때,

  실행 중 A 트랜잭션이 b에 접근해야 하고, B 트랜잭션이 a에 접근해야 하면

  a, b 모두 잠금이 설정돼 트랜잭션에서 영원히 해당 레코드에 접근할 수 없는 **교착 상태가 발생**할 수 있음

비즈니스 로직

리포지토리

엔티티 클래스를 참고하여 데이터의 CRUD를 처리하는 인터페이스로, 그 구현체는 Spring이 자동으로 생성함

```java
public interface StockRepository extends JpaRepository<Stock, Long> {
    @Lock(LockModeType.PESSIMISTIC_WRITE)
    @Query("select s from Stock s where s.id = :id")
    Stock findByIdWithPessimisticLock(Long id);
}
```

실제 데이터베이스에 전송되는 쿼리

```
SELECT * FROM stock WHERE stock.id = ? FOR UPDATE;
```

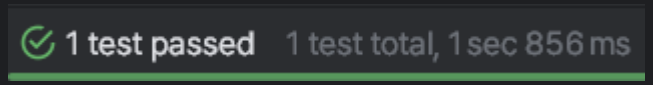
서비스

사용자의 요청이 오면 비즈니스 로직을 처리하는 Java 클래스 및 Spring 빈 컨테이너가 생성된 인스턴스가 참조

```java
@Transactional
public void decrease(Long id, Long quantity) {
    Stock stock = stockRepository.findByIdWithPessimisticLock(id);

    stock.decrease(quantity);

    stockRepository.saveAndFlush(stock);
}
```



비관적 잠금의 특징

트랜잭션 간 충돌 예방

레코드를 잠가 동시에 데이터를 수정할 때 발생할 수 있는 충돌을 방지할 수 있음

데이터 무결성 보장

동시 접근으로 발생할 수 있는 데이터 정합성 문제를 차단

성능 저하

잠금이 설정되면 다른 트랜잭션이 대기해야 하므로 잠금이 길어져 대기하는 트랜잭션이 많아지는 병목 현상 발생 가능

교착 상태 발생 가능

두 트랜잭션이 각 트랜잭션에서 필요한 서로 다른 자원을 잠가 대기할 수 있는 교착 상태 발생 가능

04

낙관적 잠금

낙관적 잠금 *(Optimistic Lock)*

서버 1	Stock	서버 2
`select * from stock where id = 1`	`{id: 1, quantity: 100, version: 1}`	read `{id: 1, quantity: 100, version: 1}`
`update quantity = 98, version + 1 from stock where id = 1 and version = 1`	`{id: 1, quantity: 98, version: 2}`	
	`{id: 1, quantity: 98, version: 2}`	`update quantity = 98, version + 1 from stock where id = 1 and version = 1` 실패!

- 데이터베이스에 잠금을 걸지 않고 버저닝*(Versioning)*을 통해 정합성을 유지할 수 있음
  - 트랜잭션에서 데이터를 읽었을 때와 갱신 수행 시의 버전이 일치하면 갱신하고, 그렇지 않으면 재시도

- 비즈니스 로직 외에도 재시도 로직을 개발자가 추가로 작성해 주어야 함
- 충돌이 빈번하게 발생하는 로직의 경우 비관적 잠금을 사용하는 것이 권장됨

비즈니스 로직

id	product_id	quantity	version
주요 키 *(레코드를 유일하게 식별할 수 있는 값)*	상품 외래 키 *(다른 테이블과 연결을 위한 다른 테이블의 주요 키)*	상품 수량	버전

엔티티 데이터베이스 테이블에 저장된 레코드*(튜플)*과 연결되는 Java 객체

```java
@Entity
public class Stock {
    // id, productId, quantity

    @Version
    private Long version;
}
```

리포지토리 엔티티 클래스를 참고하여 데이터의 CRUD를 처리하는 인터페이스로, 그 구현체는 Spring이 자동으로 생성함

```java
public interface StockRepository extends JpaRepository<Stock, Long> {
    @Lock(LockModeType.OPTIMISTIC)
    @Query("select s from Stock s where s.id = :id")
    Stock findByIdWithOptimisticLock(Long id);
}
```

비즈니스 로직

서비스 사용자의 요청이 오면 비즈니스 로직을 처리하는 Java 클래스 및 Spring 빈 컨테이너가 생성된 인스턴스가 참조

```java
public void decrease(Long id, Long quantity) throws InterruptedException {
    while (true) {
        try {
            optimisticLockStockService.decrease(id, quantity);
            break;
        } catch (Exception e) {
            Thread.sleep(50);
        }
    }
}
```

버전이 달라지면 일치하는 레코드를 찾을 수 없게 되므로
50ms 대기 후 재시도

```
1 test passed  1 test total, 5 sec 175 ms
Hibernate: select s1_0.id,s1_0.product_id,s1_0.quantity,s1_0.version from stock s1_0 where s1_0.id=?
Hibernate: update stock set product_id=?,quantity=?,version=? where id=? and version=?
```

낙관적 잠금의 특징

비교적 성능 이점

비관적 잠금과 다르게 데이터베이스 잠금 전략을 활용하지 않아 충돌이 적은 상황에서 비교적 빠른 처리가 가능

데이터 무결성 보장

동시 접근으로 발생할 수 있는 데이터 정합성 문제를 차단

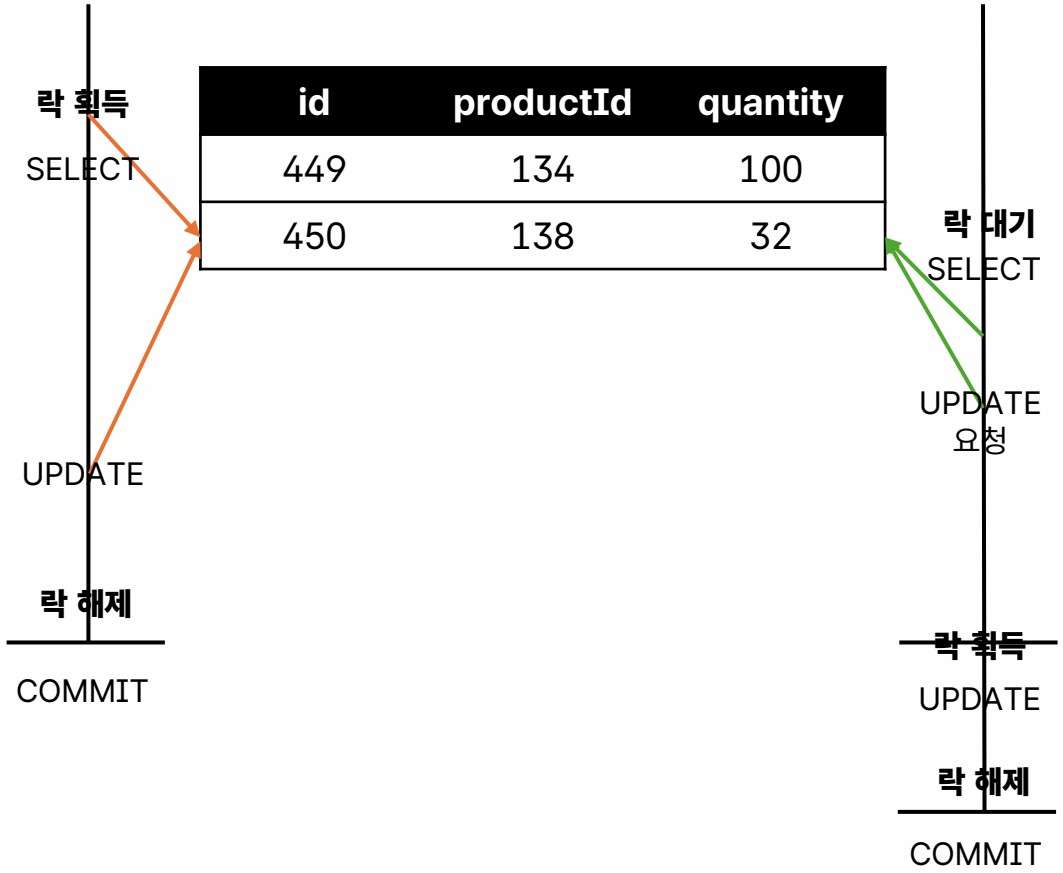
무한 재시도 발생 가능

충돌이 지속적으로 발생하면 무한히 시도할 수 있어 상한을 정해야 함

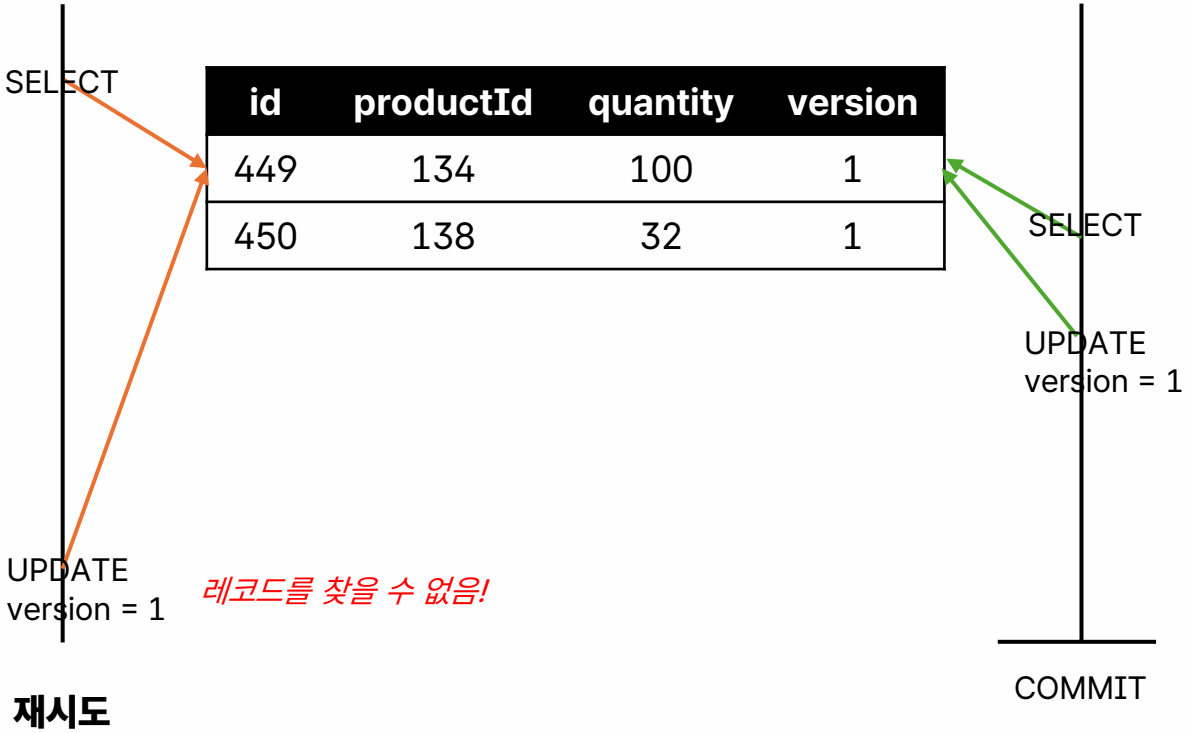
05

정리

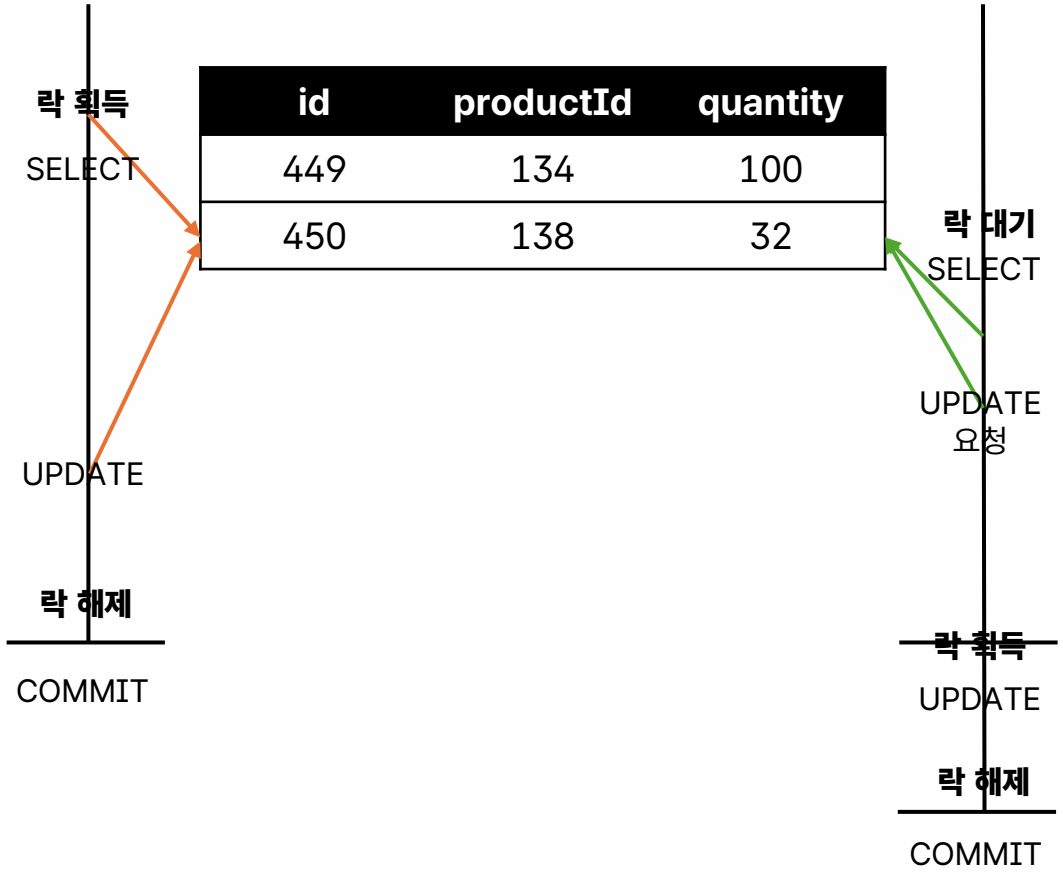
비관적 잠금


락 획득
SELECT
UPDATE
락 해제
COMMIT
id
productId
quantity
449
134
100
450
138
32
락 대기
SELECT
UPDATE
요청
락 획득
UPDATE
락 해제
COMMIT

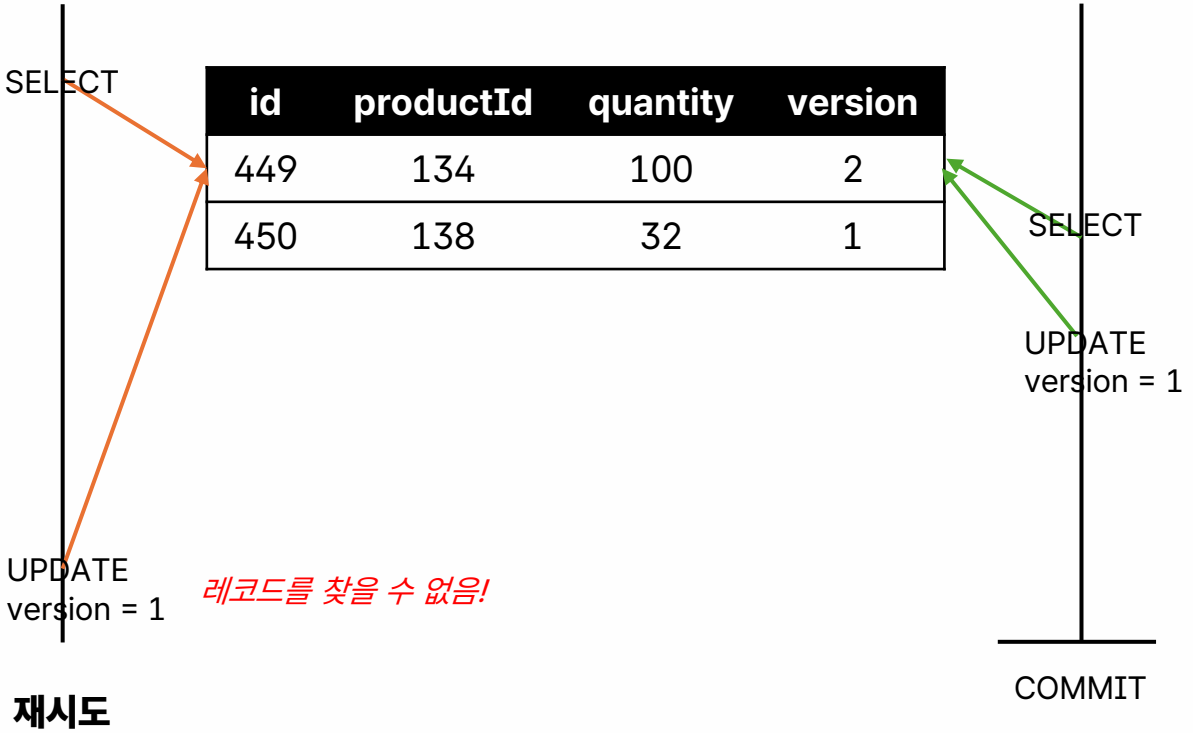
낙관적 잠금


SELECT
UPDATE
version = 1
재시도
레코드를 찾을 수 없음!
id
productId
quantity
version
449
134
100
1
450
138
32
1
SELECT
UPDATE
version = 1
COMMIT

비관적 잠금


락 획득
SELECT
UPDATE
락 해제
COMMIT
id
productId
quantity
449
134
100
450
138
32
락 대기
SELECT
UPDATE
요청
락 획득
UPDATE
락 해제
COMMIT

낙관적 잠금


SELECT
UPDATE
version = 1
재시도
레코드를 찾을 수 없음!
id
productId
quantity
version
449
134
100
2
450
138
32
1
SELECT
UPDATE
version = 1
COMMIT

비관적 잠금과 낙관적 잠금 사용 시나리오

비관적 잠금 트랜잭션 간 충돌이 빈번하다고 예상될 때 사용

은행 계좌 이체 시스템

두 명의 고객이 동시에 같은 계좌에서 돈을 출금 시 잔액이 잘못 계산될 수 있는 상황

콘서트 좌석 예약 시스템

여러 명의 고객이 같은 좌석에 접근하며 최종 결제자 외의 사람은 실패해 UX가 좋지 못할 수 있는 상황

낙관적 잠금 트랜잭션 간 충돌이 빈번하지 않을 것이라 예상될 때 사용

게시물 좋아요 시스템

좋아요 일부가 누락되거나 재시도해도 무방한 상황

들어 주셔서 감사합니다

E-mail	park@duck.com
LinkedIn	yeonjong-park
Instagram	yeonjong.park
GitHub	patulus

2025.08.20.

System Software Lab.