트랜잭션과 락

트랜잭션 기초와 JPA가 제공하는 낙관적 락과 비관적 락에 대해 알아보자.

트랜잭션과 격리 수준

트랜잭션은 ACID[1]라 하는 원자성(Atomicity), 일관성(Consistency), 격리성(Isolation), 지속성(Durability)을 보장해야 한다.

- 원자성: 트랜잭션 내에서 실행한 작업들은 마치 하나의 작업인 것처럼 모두 성공하든가 모두 실패해야 한다.
- 일관성: 모든 트랜잭션은 일관성 있는 데이터베이스 상태를 유지해야 한다. 예를 들어 데이터베이스에서 정한 무결성 제약 조건을 항상 만족해야 한다.
- 격리성: 동시에 실행되는 트랜잭션들이 서로에게 영향을 미치지 않도록 격리한다. 예를 들어 동시에 같은 데이터를 수정하지 못하도록 해야 한다. 격리성은 동시성과 관련된 성능 이슈로 인해 격리 수준을 선택할 수 있다.
- 지속성: 트랜잭션을 성공적으로 끝내면 그 결과가 항상 기록되어야 한다. 중간에 시스템에 문제가 발생해도 데이터베이스 로그 등을 사용해서 성공한 트랜잭션 내용을 복구해야 한다.

트랜잭션은 원자성, 일관성, 지속성을 보장한다. 문제는 격리성인데 트랜잭션 간에 격리성을 완벽히 보장하려면 트랜잭션을 거의 차례대로 실행해야 한다. 이렇게 하면 동시성 처리 성능이 매우 나빠진다. 이런 문제로 인해 ANSI 표준은 트랜잭션의 격리 수준을 4단계로 나누어 정의했다.

트랜잭션 격리 수준(isolation level)

- READ UNCOMMITED (커밋되지 않은 읽기)
- READ COMMITTED (커밋된 읽기)
- REPEATABLE READ (반복 가능한 읽기)
- SERIALIZABLE (직렬화 가능)

순서대로 READ UNCOMMITED의 격리 수준이 가장 낮고 SERIALIZABLE의 격리 수준이 가장 높다. 격리 수준이 낮을수록 동시성은 증가하지만, 격리 수준에 따른 다양한 문제가 발생한다.

테이블 - 트랜잭션 격리 수준과 문제점

격리 수준	DIRTY READ	NON-REPEATABLE READ	PHANTOM READ
READ UNCOMMITTED	O	O	O
READ COMMITTED		O	O
REPEATABLE READ			O
SERIALIZABLE			

격리 수준에 따른 문제점

- DIRTY READ
- NON-REPEATABLE READ(반복 불가능한 읽기)
- PHANTOM READ

격리 수준이 낮을수록 더 많은 문제가 발생한다. 트랜잭션 격리 수준에 따른 문제점을 알아보자.

- **READ UNCOMMITTED:** 커밋 하지 않은 데이터를 읽을 수 있다. 예를 들어 트랜잭션1이 데이터를 수정하고 있는데 커밋하지 않아도 트랜잭션2가 수정 중인 데이터를 조회할 수 있다. 이것을 DIRTY READ라 한다. 트랜잭션2가 DIRTY READ한 데이터를 사용하는데 트랜잭션1을 롤백하면 데이터 정합성에 심각한 문제가 발생할 수 있다. DIRTY READ를 허용하는 격리 수준을 READ UNCOMMITTED라 한다.

- **READ COMMITTED:** 커밋한 데이터만 읽을 수 있다. 따라서 DIRTY READ가 발생하지 않는다. 하지만 NON-REPEATABLE READ는 발생할 수 있다. 예를 들어 트랜잭션1이 회원 A를 조회 중인데 갑자기 트랜잭션2가 회원 A를 수정하고 커밋하면 트랜잭션1이 다시 회원 A를 조회했을 때 수정된 데이터가 조회된다. 이처럼 반복해서 같은 데이터를 읽을 수 없는 상태를 NON-REPEATABLE READ라 한다. DIRTY READ는 허용하지 않지만, NON-REPEATABLE READ는 허용하는 격리 수준을 READ COMMITTED라 한다.

- **REPEATABLE READ:** 한번 조회한 데이터를 반복해서 조회해도 같은 데이터가 조회된다. 하지만 PHANTOM READ는 발생할 수 있다. 예를 들어 트랜잭션1이 10살 이하의 회원을 조회했는데 트랜잭션2가 5살 회원을 추가하고 커밋하면 트랜잭션1이 다시 10살 이하의 회원을 조회했을 때 회원 하나가 추가된 상태로 조회된다. 이처럼 반복 조회 시 결과 집합이 달라지는 것을 PHANTOM READ라 한다. NON-REPEATABLE READ는 허용하지 않지만, PHANTOM READ는 허용하는 격리 수준을 REPEATABLE READ라 한다.

- **SERIALIZABLE:** 가장 엄격한 트랜잭션 격리 수준이다. 여기서는 PHANTOM READ가 발생하지 않는다. 하지만 동시성 처리 성능이 급격히 떨어질 수 있다.

애플리케이션 대부분은 동시성 처리가 중요하므로 데이터베이스들은 보통 READ COMMITTED 격리 수준을 기본으로 사용한다. 일부 중요한 비즈니스 로직에 더 높은 격리 수준이 필요하면 데이터베이스 트랜잭션이 제공하는 잠금 기능을 사용하면 된다.

> **참고:** 트랜잭션 격리 수준에 따른 동작 방식은 데이터베이스마다 조금씩 다르다. 최근에는 데이터베이스들이 더 많은 동시성 처리를 위해 락보다는 MVCC[2](Multiversion concurrency control)를 사용하므로 락을 사용하는 데이터베이스와 약간 다른 특성을 지닌다.

낙관적 락과 비관적 락 기초

JPA의 영속성 컨텍스트(1차 캐시)를 적절히 활용하면 데이터베이스 트랜잭션이 READ COMMITTED 격리 수준이어도 애플리케이션 레벨에서 반복 가능한 읽기(REPEATABLE READ)가 가능하다. 물론 엔티티가 아닌 스칼라 값을 직접 조회하면 영속성 컨텍스트의 관리를 받지 못하므로 반복 가능한 읽기를 할 수 없다.

JPA는 데이터베이스 트랜잭션 격리 수준을 READ COMMITTED 정도로 가정한다. 만약 더 높은 격리 수준이 필요하면 낙관적 락과 비관적 락 중 하나를 사용하면 된다.

낙관적 락은 이름 그대로 트랜잭션 대부분은 충돌이 발생하지 않는다고 낙관적으로 가정하는 방법이다. 이것은 데이터베이스가 제공하는 락 기능을 사용하는 것이 아니라 JPA가 제공하는 버전 관리 기능을 사용한다. 쉽게 이야기해서 애플리케이션이 제공하는 락이다. 낙관적 락은 트랜잭션을 커밋하기 전까지는 트랜잭션의 충돌을 알 수 없다는 특징이 있다.

비관적 락은 이름 그대로 트랜잭션의 충돌이 발생한다고 가정하고 우선 락을 걸고 보는 방법이다. 이것은 데이터베이스가 제공하는 락 기능을 사용한다. 대표적으로 `select for update` 구문이 있다.

second lost updates problem(두 번의 갱신 분실 문제)

여기에 추가로 데이터베이스 트랜잭션 범위를 넘어서는 문제도 있다. 예를 들어 사용자 A와 B가 동시에 제목이 같은 공지사항을 수정한다고 생각해보자. 둘이 동시에 수정 화면을 열어서 내용을 수정하는 중에 사용자 A가 먼저 수정완료 버튼을 눌렀다. 잠시후에 사용자 B가 수정완료 버튼을 눌렀다. 결과적으로 먼저 완료한 사용자A의 수정사항은 사라지고 나중에 완료한 사용자B의 수정사항만 남게된다.

이런 문제는 데이터베이스 트랜잭션의 범위를 넘어선다. 따라서 트랜잭션만으로는 문제를 해결할 수 없다.

이때는 3가지 선택 방법이 있다.

- 마지막 커밋만 인정하기: 사용자 A의 내용은 무시하고 마지막에 커밋한 사용자 B의 내용만 인정한다.
- 최초 커밋만 인정하기: 사용자 A가 이미 수정을 완료했으므로 사용자 B가 수정을 완료할 때 오류가 발생한다.
- 충돌하는 갱신 내용 병합하기: 사용자 A와 사용자 B의 수정사항을 병합한다.

기본은 마지막 커밋만 인정하기가 사용된다. 하지만 상황에 따라 최초 커밋만 인정하기가 더 합리적일 수 있다. JPA가 제공하는 버전 관리 기능을 사용하면 손쉽게 최초 커밋만 인정하기를 구현할 수 있다. 충돌하는 갱신 내용 병합하기는 최초 커밋만 인정하기를 조금 더 우아하게 처리하는 방법인데 애플리케

이션 개발자가 직접 사용자를 위해 병합 방법을 제공해야 한다.

@Version

JPA가 제공하는 낙관적 락을 사용하려면 `@Version` 어노테이션을 사용해서 버전 관리 기능을 추가해야 한다.

@Version 적용 가능 타입

- `Long` (`long`)
- `Integer` (`int`)
- `Short` (`short`)
- `Timestamp`

====== 엔티티에 버전 관리 추가 ======

```
@Entity
public class Board {

    @Id
    private String id;
    private String title;

    @Version //**
    private Integer version;
}
```

버전 관리 기능을 적용하려면 엔티티에 버전 관리용 필드를 하나 추가하고 `@Version`을 붙이면 된다. 이제부터 엔티티를 수정할 때 마다 버전이 하나씩 증가한다. 그리고 엔티티를 수정할 때 조회 시점의 버전과 수정 시점의 버전이 다르면 예외가 발생한다. 예를 들어 조회한 엔티티를 수정하고 있는데 다른 트랜잭션에서 같은 엔티티를 수정하고 커밋해서 버전이 증가해버리면 트랜잭션을 커밋할 때 버전 정보가 다르므로 예외가 발생한다.

다음 예제 코드와 [그림 - Version]을 통해 자세히 알아보자.

```
Board board = em.find(Board.class, id); //트랜잭션1 조회 title="제목A", version=1

//트랜잭션2에서 해당 게시물을 수정해서 title="제목C", version=2로 증가

board.setTitle("제목B"); //트랜잭션1 데이터 수정
```

```
save(board);
tx.commit(); //예외 발생, 데이터베이스 version=2, 엔티티 version=1
```

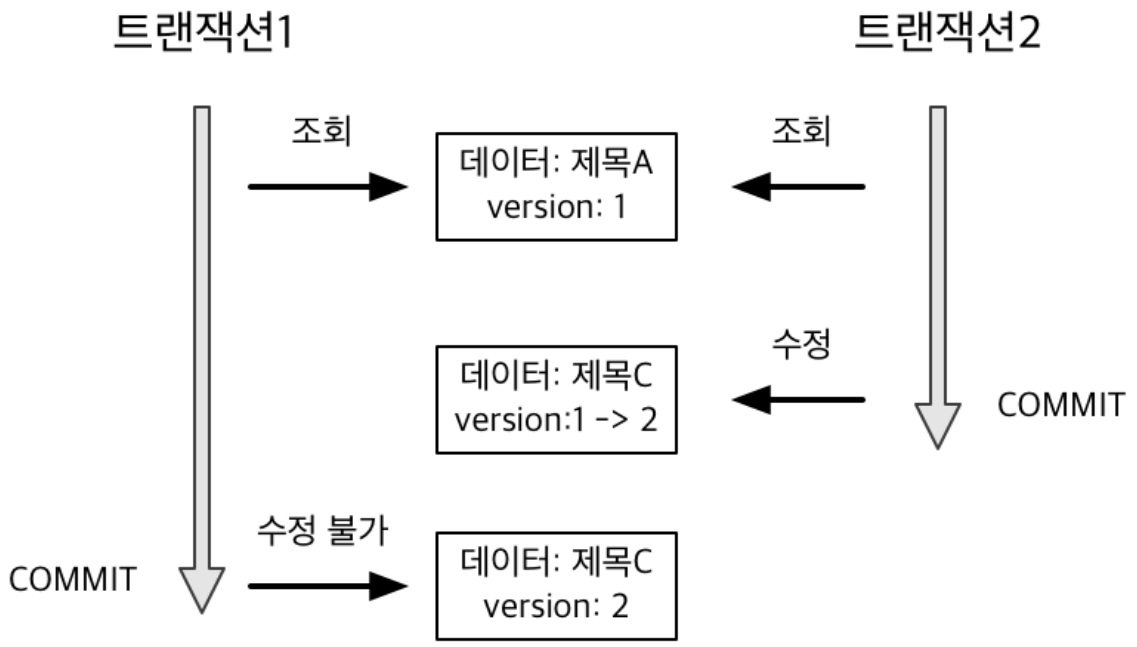

그림 - Version

제목이 A고 버전이 1인 게시물이 있다. 트랜잭션1은 이것을 제목 B로 변경하려고 조회했다. 이때 트랜잭션2가 해당 데이터를 조회해서 제목을 C로 수정하고 커밋해서 버전 정보가 2로 증가했다. 이후 트랜잭션1이 데이터를 제목 B로 변경하고 트랜잭션을 커밋하는 순간 엔티티를 조회할 때 버전과 데이터베이스의 현재 버전 정보가 다르므로 예외가 발생한다. 따라서 **버전 정보를 사용하면 최초 커밋만 인정**하기가 적용된다.

- 버전 정보 비교 방법

JPA가 버전 정보를 비교하는 방법은 단순하다. 엔티티를 수정하고 트랜잭션을 커밋하면 영속성 컨텍스트를 플러시 하면서 UPDATE 쿼리를 실행한다. 이때 버전을 사용하는 엔티티면 검색 조건에 엔티티의 버전 정보를 추가한다.

====== 버전 사용 엔티티 SQL ======

```
UPDATE BOARD
SET
    TITLE=?,
    VERSION=? (버전 + 1 증가) //**
WHERE
```

```
    ID=?
    AND VERSION=? (버전 비교) //**
```

데이터베이스 버전과 엔티티 버전이 같으면 데이터를 수정하면서 동시에 버전도 하나 증가시킨다. 만약 데이터베이스에 버전이 이미 증가해서 수정중인 엔티티의 버전과 다르면 UPDATE 쿼리의 WHERE 문에서 VERSION 값이 다르므로 수정할 대상이 없다. 이때는 버전이 이미 증가한 것으로 판단해서 JPA가 예외를 발생시킨다.

버전 증가

버전은 엔티티의 값을 변경하면 증가한다. 그리고 값 타입인 임베디드 타입과 값 타입 컬렉션은 논리적인 개념상 해당 엔티티의 값이므로 수정하면 엔티티의 버전이 증가한다. 단 연관관계 필드는 외래 키를 관리하는 연관관계의 주인 필드를 수정할 때만 버전이 증가한다.

`@Version`으로 추가한 버전 관리 필드는 JPA가 직접 관리하므로 개발자가 임의로 수정하면 안 된다. (벌크 연산 제외) 만약 버전 값을 강제로 증가하려면 특별한 락 옵션을 선택하면 된다. 락 옵션은 조금 뒤에 알아보자.

> **참고:** 벌크 연산은 버전을 무시한다. 벌크 연산에서 버전을 증가하려면 버전 필드를 강제로 증가시켜야 한다.
>
> ```
> update Member m set m.name = '변경', m.version = m.version + 1
> ```

JPA 락 사용하기

JPA가 제공하는 락을 어떻게 사용하는지 알아보자.

> **참고:** JPA를 사용할 때 추천하는 전략은 READ COMMITTED 트랜잭션 격리 수준 + 낙관적 버전 관리다.(두 번의 갱신 내역 분실 문제 예방)

락은 다음 위치에 적용할 수 있다.

- `EntityManager.lock()`, `EntityManger.find()`, `EntityManager.refresh()`
- `Query.setLockMode()` (`TypeQuery` 포함)

- `@NamedQuery`

다음처럼 조회하면서 즉시 락을 걸 수도 있고

```
Board board = em.find(Board.class, id, LockModeType.OPTIMISTIC);
```

다음처럼 필요할 때 락을 걸 수도 있다.

```
Board board = em.find(Board.class, id);
..
em.lock(board, LockModeType.OPTIMISTIC);
```

JPA가 제공하는 락 옵션은 `javax.persistence.LockModeType`에 정의되어 있다.

`LockModeType` 속성

락 모드	타입	설명
낙관적 락	OPTIMISTIC	낙관적 락을 사용한다.
낙관적 락	OPTIMISTIC_FORCE_INCREMENT	낙관적 락 + 버전정보를 강제로 증가한다.
비관적 락	PESSIMISTIC_READ	비관적 락, 읽기 락을 사용한다.
비관적 락	PESSIMISTIC_WRITE	비관적 락, 쓰기 락을 사용한다.
비관적 락	PESSIMISTIC_FORCE_INCREMENT	비관적 락 + 버전정보를 강제로 증가한다.
기타	NONE	락을 걸지 않는다.
기타	READ	JPA1.0 호환 기능이다. `OPTIMISTIC`과 같으므로 `OPTIMISTIC`을 사용하면 된다.
기타	WRITE	JPA1.0 호환 기능이다. `OPTIMISTIC_FORCE_IMCREMENT`와 같다.

JPA 낙관적 락

JPA가 제공하는 낙관적 락은 버전(`@Version`)을 사용한다. 따라서 낙관적 락을 사용하려면 버전이 있어야 한다.

낙관적 락은 트랜잭션을 커밋하는 시점에 충돌을 알 수 있다는 특징이 있다.

> **참고:** 일부 JPA 구현체 중에는 `@Version` 컬럼 없이 낙관적 락을 허용하기도 하지만 추천하지는 않는다.

참고로 락 옵션 없이 `@Version` 만 있어도 낙관적 락이 적용된다. 락 옵션을 사용하면 락을 더 세밀하게 제어할 수 있다. 낙관적 락의 옵션에 따른 효과를 하나씩 알아보자.

- NONE

락 옵션을 적용하지 않아도 엔티티에 `@Version` 이 적용된 필드만 있으면 낙관적 락이 적용된다. 자세한 내용은 앞의 `@Version` 에서 이미 설명했다.

- 용도: 조회한 엔티티를 수정할 때 다른 트랜잭션에 의해 변경(삭제)되지 않아야 한다. 조회 시점부터 수정 시점까지를 보장한다.
- 동작: 엔티티를 수정할 때 버전을 체크하면서 버전을 증가한다. (UPDATE 쿼리 사용) 이때 데이터베이스의 버전 값이 현재 버전이 아니면 예외가 발생한다.
- 이점: 두 번의 갱신 분실 문제(second lost updates problem)을 예방한다.
- 예외
  - `javax.persistence.OptimisticLockException` (JPA 예외)
  - `org.hibernate.StaleObjectStateException` (하이버네이트 예외)
  - `org.springframework.orm.ObjectOptimisticLockingFailureException` (스프링 예외 추상화)

- OPTIMISTIC

`@Version` 만 적용했을 때는 엔티티를 수정해야 버전을 체크하지만 이 옵션을 추가하면 엔티티를 조회만 해도 버전을 체크한다. 쉽게 이야기해서 한번 조회한 엔티티는 트랜잭션을 종료할 때까지 다른 트랜잭션에서 변경하지 않음을 보장한다.

- 용도: 조회한 엔티티는 트랜잭션이 끝날 때까지 다른 트랜잭션에 의해 변경되지 않아야 한다. 조회 시점부터 트랜잭션이 끝날 때까지 조회한 엔티티가 변경되지 않음을 보장한다.
- 동작: 트랜잭션을 커밋할 때 버전 정보를 조회 해서(SELECT 쿼리 사용) 현재 엔티티의 버전과 같은지 검증한다. 만약 같지 않으면 예외가 발생한다.
- 이점: OPTIMISTIC 옵션은 DIRTY READ와 NON-REPEATABLE READ를 방지한다.
- 예외
  - `javax.persistence.OptimisticLockException` (JPA 예외)

- `org.hibernate.StaleObjectStateException` (하이버네이트 예외)
- `org.springframework.orm.ObjectOptimisticLockingFailureException` (스프링 예외 추상화)

다음 예제를 통해 알아보자.

```
//트랜잭션1 조회 title="제목A", version=1
Board board = em.find(Board.class, id, LockModeType.OPTIMISTIC);

//중간에 트랜잭션2에서 해당 게시물을 수정해서 title="제목C", version=2로 증가

//트랜잭션1 커밋 시점에 버전 정보 검증, 예외 발생(데이터베이스 version=2, 엔티티 version=1)
tx.commit();
```

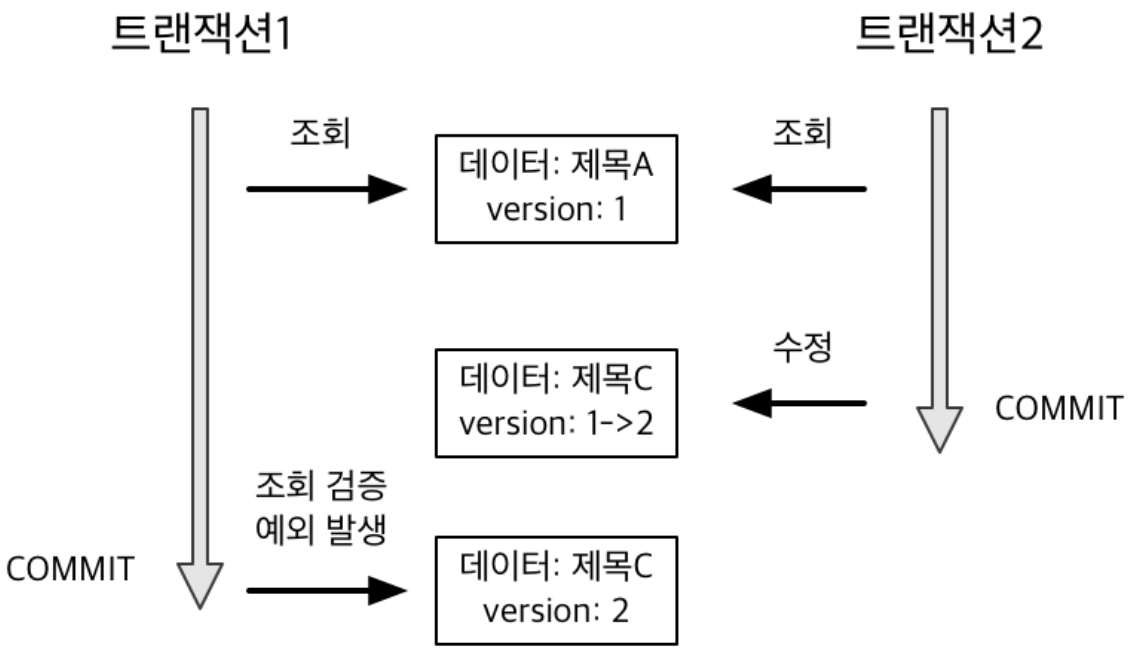

그림 - OPTIMISTIC

[그림 - OPTIMISTIC]을 보면 트랜잭션1은 `OPTIMISTIC` 락으로 버전이 1인 데이터를 조회했다. 이후에 트랜잭션2가 데이터를 수정해버렸고 버전은 2로 증가했다. 트랜잭션1은 엔티티를 `OPTIMISTIC` 락으로 조회했으므로 트랜잭셕을 커밋할 때 데이터베이스에 있는 버전 정보를 **SELECT 쿼리로 조회해서 처음에 조회한 엔티티의 버전 정보와 비교한다.** 이때 버전 정보가 다르면 예외가 발생한다.

락을 걸지 않고 `@Version` 만 사용하면 엔티티를 수정해야 버전 정보를 확인하지만 `OPTIMISTIC` 옵션을 사용하면 **수정하지 않고 단순히 조회만 해도 버전을 확인한다.**

- OPTIMISTIC_FORCE_INCREMENT

낙관적 락을 사용하면서 버전 정보를 강제로 증가한다.

- 용도: 논리적인 단위의 엔티티 묶음을 관리할 수 있다.
- 예제: 게시물과 첨부파일이 일대다, 다대일의 양방향 연관관계고 첨부파일이 연관관계의 주인이다. 게시물을 수정하는데 단순히 첨부파일만 추가하면 게시물의 버전은 증가하지 않는다. 해당 게시물은 물리적으로는 변경되지 않았지만, 논리적으로는 변경되었다. 이때 게시물의 버전도 강제로 증가하려면 `OPTIMISTIC_FORCE_INCREMENT`를 사용하면 된다.
- 동작: 엔티티를 수정하지 않아도 트랜잭션을 커밋할 때 UPDATE 쿼리를 사용해서 버전 정보를 강제로 증가시킨다. 이때 데이터베이스의 버전이 엔티티의 버전과 다르면 예외가 발생한다. 추가로 엔티티를 수정하면 수정시 버전 UPDATE가 발생한다. 따라서 총 2번의 버전 증가가 나타날 수 있다.
- 이점: 강제로 버전을 증가해서 논리적인 단위의 엔티티 묶음을 버전 관리할 수 있다.
- 예외
  - `javax.persistence.OptimisticLockException` (JPA 예외)
  - `org.hibernate.StaleObjectStateException` (하이버네이트 예외)
  - `org.springframework.orm.ObjectOptimisticLockingFailureException` (스프링 예외 추상화)

> **참고:** OPTIMISTIC_FORCE_INCREMENT는 Aggregate Root[3](DDD)에 사용할 수 있다. 예를 들어 Aggregate Root는 수정하지 않았지만 Aggregate Root가 관리하는 엔티티를 수정했을 때 Aggregate Root의 버전을 강제로 증가시킬 수 있다.

예제 코드를 보자.

```
//트랜잭션1 조회 title="제목A", version=1
Board board = em.find(Board.class, id, LockModeType.OPTIMISTIC_FORCE_INCREMENT);

//트랜잭션1 커밋 시점에 버전 강제 증가
tx.commit();
```

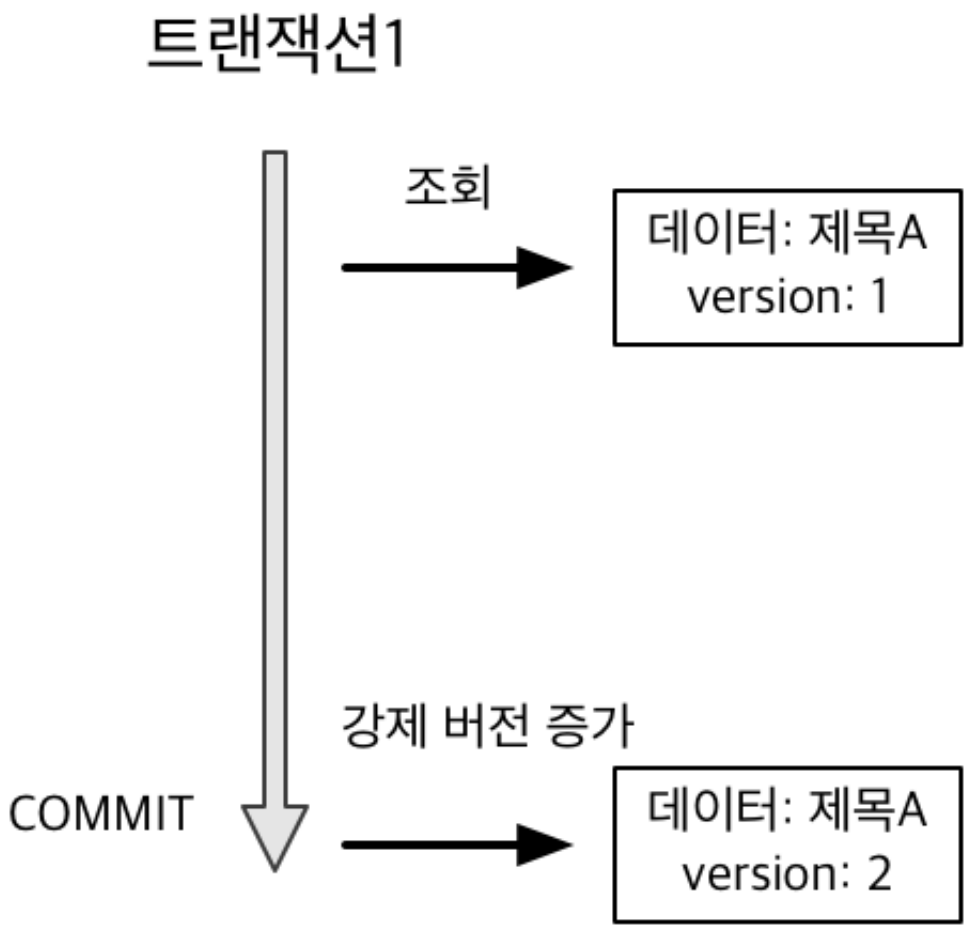

그림 - OPTIMISTIC_FORCE_INCREMENT

[그림 - OPTIMISTIC_FORCE_INCREMENT]를 보면 데이터를 수정하지 않아도 트랜잭션을 커밋할 때 버전 정보가 증가한다.

JPA 비관적 락

JPA가 제공하는 비관적 락은 데이터베이스 트랜잭션 락 메커니즘에 의존하는 방법이다. 주로 SQL 쿼리에 `select for update` 구문을 사용하면서 시작하고 버전 정보는 사용하지 않는다. 비관적 락은 주로 `PESSIMISTIC_WRITE` 모드를 사용한다.

비관적 락은 다음과 같은 특징이 있다.

- 엔티티가 아닌 스칼라 타입을 조회할 때도 사용할 수 있다.
- 데이터를 수정하는 즉시 트랜잭션 충돌을 감지할 수 있다.

- PESSIMISTIC_WRITE

비관적 락이라 하면 일반적으로 이 옵션을 뜻한다. 데이터베이스에 쓰기 락을 걸 때 사용한다.

- 용도: 데이터베이스에 쓰기 락을 건다.
- 동작: 데이터베이스 `select for update`를 사용해서 락을 건다.
- 이점: NON-REPEATABLE READ를 방지한다. 락이 걸린 로우는 다른 트랜잭션이 수정할 수 없다.
- 예외
  - `javax.persistence.PessimisticLockException` (JPA 예외)
  - `org.springframework.dao.PessimisticLockingFailureException` (스프링 예외 추상화)

- PESSIMISTIC_READ

데이터를 반복 읽기만 하고 수정하지 않는 용도로 락을 걸 때 사용한다. 일반적으로 잘 사용하지 않는다. 데이터베이스 대부분은 방언에 의해 `PESSIMISTIC_WRITE`로 동작한다.

- MySQL: `lock in share mode`
- PostgreSQL: `for share`
- 예외
  - `javax.persistence.PessimisticLockException` (JPA 예외)
  - `org.springframework.dao.PessimisticLockingFailureException` (스프링 예외 추상화)

- PESSIMISTIC_FORCE_INCREMENT

비관적 락중 유일하게 버전 정보를 사용한다. 비관적 락이지만 버전 정보를 강제로 증가시킨다. 하이버네이트는 `nowait`를 지원하는 데이터베이스에 대해서 `for update nowait` 옵션을 적용한다.

- ORACLE: `for update nowait`
- PostgreSQL: `for update nowait`
- `nowait`를 지원하지 않으면 `for update`가 사용된다.
- 예외
  - `javax.persistence.PessimisticLockException` (JPA 예외)
  - `org.springframework.dao.PessimisticLockingFailureException` (스프링 예외 추상화)

비관적 락과 타임아웃

비관적 락을 사용하면 락을 획득할 때까지 트랜잭션이 대기한다. 무한정 기다릴 수는 없으므로 타임아웃 시간을 줄 수 있다. 다음 예제는 10초간 대기해서 응답이 없으면 `javax.persistence.LockTimeoutException` 예외가 발생한다.

```
Map<String,Object> properties = new HashMap<String,Object>();
properties.put("javax.persistence.lock.timeout", 10000); //타임아웃 10초까지 대기
Board board = em.find(Board.class, "boardId", LockModeType.PESSIMISTIC_WRITE, pr
```

타임아웃은 데이터베이스 특성에 따라 동작하지 않을 수 있다.

---

1. http://en.wikipedia.org/wiki/ACID ↩
2. http://en.wikipedia.org/wiki/Multiversion_concurrency_control ↩
3. http://martinfowler.com/bliki/DDD_Aggregate.html ↩