BACKEND CS · STUDY GUIDE

동시성·트랜잭션 10강

ACID · 격리 수준 · MVCC · 락 · 멱등성

10강 · 3부 구성

ttukttak-coding.dev

2026

이 책은

이 책은 트랜잭션과 동시성 제어를 한 권에 모은 학습서입니다. ACID 원칙에서 시작해 격리 수준, MVCC, 락, 2단계 로킹, FOR UPDATE, 분산 락, 멱등성까지 — 동시 요청이 데이터를 망가뜨리지 않게 하는 모든 도구를 다룹니다.

각 강의는 "이 개념이 없으면 어디서 사고가 나는가?" 라는 질문에서 출발합니다. 단순 정의 암기가 아니라, 락 경합·팬텀 리드·중복 결제처럼 실제 운영에서 만나는 문제를 도구로 어떻게 막는지를 짚습니다.

1~3강은 트랜잭션 기초, 4~6강은 동시성 제어 메커니즘, 7~10강은 락과 멱등성의 실무 적용입니다. 순서대로 읽으시면 자연스럽게 한 줄로 이해됩니다.

> **구성**: 3부 10강 / **대상**: 트랜잭션·락이 처음이거나 다시 정리하고 싶은 백엔드 개발자 / **블로그**: 더 많은 글은 ttukttak-coding.dev 에서 보실 수 있습니다.

목차

1부. 트랜잭션 기초

2부. 동시성 제어 메커니즘

3부. 락과 멱등성 실무

PART 1

1부. 트랜잭션 기초

Chapter 1

트랜잭션 ACID 원칙 완전 정복 — `Atomicity`, `Consistency`, `Isolation`, `Durability`를 오해 없이 이해하기

#트랜잭션

ACID 네 가지 속성이 각각 무엇을 보장하고 무엇은 보장하지 않는지, MySQL InnoDB 기준 구현 포인트와 함께 정리합니다.

ACID, 왜 알아야 하나요?

트랜잭션을 쓴다고 말할 때 정말 중요한 것은 `BEGIN`과 `COMMIT` 문법이 아닙니다. **장애와 동시 요청이 들어왔을 때 무엇이 보장되고, 무엇은 직접 챙겨야 하는지**를 아는 것입니다.

다음 같은 상황에서 ACID를 정확히 이해할 필요가 있습니다.

- 계좌 이체 중 출금은 됐는데 입금 전에 예외가 발생했습니다
- 주문 저장은 성공했는데 재고 차감이나 포인트 적립은 실패했습니다
- `COMMIT` 직후 서버가 죽었는데, 방금 성공한 변경이 남아 있어야 합니다
- 같은 데이터를 여러 요청이 동시에 읽고 수정하는데 결과가 뒤엉키면 안 됩니다

ACID는 이런 상황에서 데이터베이스 트랜잭션이 어떤 신뢰성을 제공해야 하는지 설명하는 네 가지 속성입니다.

> **기준:** 이 글은 ACID의 일반 정의는 PostgreSQL 18 Glossary를 기준으로 설명하고, 구현 예시는 기존 시리즈와 맞춰 **MySQL 8.4 + `InnoDB`** 기준으로 설명합니다. `Atomicity`, `Isolation`, `Durability` 구현 포인트는 MySQL 8.4 `InnoDB and the ACID Model`, Transaction Isolation Levels, Consistent Nonlocking Reads, Optimizing InnoDB Transaction Management를 참고했습니다. `Consistency`는 관계형 데이터베이스 일반 의미인 **"커밋 시점에 유효한 상태를 만족하는가"**를 중심으로 설명하되, MySQL 문서가 `doublewrite buffer`와 crash recovery 같은 내부 보호 메커니즘도 함께 언급한다는 점을 같이 짚습니다.

먼저 가장 짧은 답부터 보면

- `Atomicity` — **전부 성공하거나 전부 실패**해야 합니다
- `Consistency` — **커밋 결과**는 무결성 규칙을 만족하는 유효한 상태여야 합니다
- `Isolation` — 동시에 실행되는 트랜잭션이 서로를 **위험하게 끼어들지 못하게** 해야 합니다
- `Durability` — 한 번 `COMMIT`된 결과는 **장애 후에도 남아 있어야** 합니다

중요한 점은 네 속성이 따로 노는 체크박스가 아니라는 점입니다. 출금과 입금을 한 덩어리로 묶는 것은 `Atomicity`, 그 결과가 음수 잔액이나 깨진

참조를 남기지 않아야 하는 것은 `Consistency`, 동시에 다른 요청이 끼어들어 중간값을 보지 않게 하는 것은 `Isolation`, 성공 응답 뒤 장애가 나도 결과가 사라지지 않아야 하는 것은 `Durability`입니다.

계좌 이체 하나만 놓고 봐도 이렇게 나눠 볼 수 있습니다.

속성	계좌 이체에서의 의미
`Atomicity`	출금과 입금이 함께 성공하거나 함께 실패해야 합니다
`Consistency`	총액 보존, 음수 잔액 금지 같은 규칙이 깨지면 안 됩니다
`Isolation`	동시에 다른 이체가 끼어들어 중간 잔액을 보고 잘못 계산하면 안 됩니다
`Durability`	이체 완료 응답 뒤 장애가 나도 결과가 사라지면 안 됩니다

Phase 1. `Atomicity` — 왜 중간 성공 상태가 남으면 안 될까요?

핵심: 한 트랜잭션은 전부 반영되거나 전부 취소되어야 합니다

PostgreSQL glossary는 원자성(`Atomicity`)을 "**모든 작업이 하나의 단위로 완료되거나 하나도 완료되지 않는 성질**"로 설명합니다. MySQL 문서도 `Atomicity`를 주로 `InnoDB` 트랜잭션과 `COMMIT` / `ROLLBACK`, `autocommit`과 연결해서 설명합니다.

문제: 출금은 됐는데 입금 전에 실패하면 데이터가 깨집니다

계좌 이체를 예로 들어 보겠습니다.

```
UPDATE account
SET balance = balance - 10000
WHERE id = 1;

-- 여기서 애플리케이션 예외 발생

UPDATE account
SET balance = balance + 10000
WHERE id = 2;
```

트랜잭션 없이 이렇게 실행하면 첫 번째 `UPDATE`는 이미 반영됐는데, 두 번째 `UPDATE`는 실행되지 않을 수 있습니다. 그러면 돈은 한쪽 계좌에서만 빠져나간 상태가 됩니다.

해결: 하나의 업무 단위를 한 트랜잭션으로 묶습니다

```
START TRANSACTION;

UPDATE account
SET balance = balance - 10000
WHERE id = 1;

UPDATE account
SET balance = balance + 10000
WHERE id = 2;

COMMIT;
```

이제 중간에 오류가 나면 `ROLLBACK` 되어 둘 다 반영되지 않습니다.

```
START TRANSACTION;

UPDATE account
SET balance = balance - 10000
WHERE id = 1;

-- 예외 발생
ROLLBACK;
```

핵심은 이것입니다.

- **업무적으로 하나여야 하는 작업은 DB에서도 하나의 단위로 커밋해야 합니다**

- 중간 단계가 밖으로 새면, 그 순간 `Atomicity`가 깨집니다

> **참고:** MySQL 기본값 `AUTOCOMMIT=1`에서는 각 SQL 문장이 자동으로 하나의 트랜잭션이 됩니다. 즉, 여러 SQL이 함께 성공하거나 함께 실패해야 하는 경우에는 **명시적으로 하나의 트랜잭션으로 묶어야** 합니다.

Phase 2. `Consistency` — 일관성은 "내가 기대한 결과"와 같은 말일까요?

핵심: 커밋 시점의 데이터는 유효한 상태를 만족해야 합니다

PostgreSQL glossary는 `Consistency`를 "**데이터가 항상 integrity constraints를 만족하는 성질**"로 설명합니다. 이 정의가 ACID를 이해할 때 가장 직관적입니다.

즉, 트랜잭션은 잠깐 중간 상태를 거칠 수 있어도, **커밋되는 최종 상태는 유효해야** 합니다.

문제: 부분 성공이 아니라, 잘못된 상태 자체가 커밋될 수 있습니다

예를 들어 잔액이 음수가 되면 안 된다고 가정하겠습니다.

```
CREATE TABLE account (
    id BIGINT PRIMARY KEY,
    balance INT NOT NULL CHECK (balance ⩾ 0)
);
```

이 상태에서 아래 트랜잭션이 실행되면:

```
START TRANSACTION;

UPDATE account
SET balance = balance - 10000
WHERE id = 1;

COMMIT;
```

잔액이 `-1000`이 된다면 `CHECK (balance ⩾ 0)`를 위반하므로, 유효하지 않은 상태를 커밋할 수 없습니다.

여기서 중요한 점은 **"언제 검사하느냐"보다 "유효하지 않은 최종 상태를 남기지 않느냐"** 입니다. 실제로는 제약 종류와 DBMS에 따라 오류가 `UPDATE` 시점에 나기도 하고, `COMMIT` 시점에 나기도 합니다. ACID 관점의 핵심은 **잘못된 상태가 성공적으로 확정되지 않는 것**입니다.

무엇이 일관성을 만들까요?

일관성은 보통 아래 규칙들이 함께 만듭니다.

- `PRIMARY KEY`, `UNIQUE`
- `NOT NULL`
- `FOREIGN KEY`
- `CHECK`
- 트랜잭션 안에서 유지해야 하는 도메인 규칙

예를 들어 주문 헤더 없이 주문 항목만 들어가면 안 된다는 규칙은 `FOREIGN KEY`가 지켜 줍니다. 같은 쿠폰을 두 번 발급하면 안 된다는 규칙은 `UNIQUE`가 도와줄 수 있습니다.

여기서 많이 하는 오해

`Consistency`는 **"결과가 내가 원한 비즈니스 결과다"**와 같은 말이 아닙니다.

예를 들어 실수로 잘못된 사용자에게 포인트를 적립했는데:

- 참조 무결성은 맞고
- 음수 값도 없고
- `UNIQUE` 위반도 없고
- 모든 SQL이 정상 커밋됐다면

이 트랜잭션은 **ACID 관점에서는 consistent할 수 있습니다.** 하지만 비즈니스는 틀렸습니다.

즉:

- **ACID의 `Consistency`** — DB가 유효한 상태를 유지하는가
- **업무적 정합성** — 애플리케이션 로직이 진짜로 올바른 대상을 처리했는가

는 겹치지만 완전히 같은 말은 아닙니다.

> **참고:** MySQL의 `InnoDB and the ACID Model` 문서는 `Consistency`를 설명할 때 `doublewrite buffer`와 crash recovery 같은 내부 보호 메커니즘을 함께 언급합니다. 이것은 "유효한 상태가 장애 후에도 깨지지 않도록 엔진이 내부적으로 보호한다"는 관점입니다. 이 글에서는 독자가 가장 많이 헷갈리는 지점인 **무결성 규칙을 만족하는 상태**를 먼저 기준으로 잡고 읽는 편이 이해가 쉽습니다.

Phase 3. `Isolation` — 동시에 실행되는데 왜 서로를 못 본다고 말할까요?

핵심: 동시에 실행되는 트랜잭션 사이의 간섭을 제어합니다

`Isolation`은 ACID의 네 속성 중 **동시성**과 가장 직접적으로 연결됩니다. MySQL 문서는 `Isolation`을 트랜잭션 격리 수준과 락, 그리고 `InnoDB`의 읽기 방식과 연결해 설명합니다.

문제는 트랜잭션이 동시에 실행될 수 있다는 점입니다.

```
트랜잭션 A                    트랜잭션 B
──────────────────────────────────────────
SELECT balance = 10000
                              UPDATE balance = 5000
                              COMMIT
SELECT balance = ?
```

두 번째 `SELECT`가 무엇을 봐야 할까요?

- 항상 최신 커밋값을 볼 수도 있고
- 처음 읽은 시점의 스냅샷을 계속 볼 수도 있고
- 경우에 따라 대기하거나 락을 잡아야 할 수도 있습니다

이 선택을 구체화한 것이 **격리 수준**입니다.

MySQL InnoDB는 어떻게 구현할까요?

MySQL 문서에 따르면 `InnoDB` 트랜잭션 모델은 **multi-versioning**과 **traditional two-phase locking**을 결합합니다.

- 일반 `SELECT`는 기본적으로 **nonlocking consistent read**로 동작합니다
- `REPEATABLE READ`에서는 트랜잭션 안의 일반 `SELECT`가 첫 읽기 시점의 스냅샷을 계속 봅니다
- `READ COMMITTED`에서는 각 읽기마다 더 새로운 스냅샷을 볼 수 있습니다
- `FOR UPDATE`, `FOR SHARE`, `UPDATE`, `DELETE`는 잠금 전략이 같이 개입합니다

즉, `Isolation`은 "무조건 락으로 다 막는다"가 아니라 **MVCC 스냅샷과 락을 조합해서 간섭을 제어하는 정책**에 가깝습니다.

그래서 실무에서는 어떻게 읽어야 할까요?

- `Isolation`은 **동시에 실행되는 트랜잭션끼리 무엇을 볼 수 있는지**의 문제입니다

- `Atomicity`가 한 트랜잭션 내부의 묶음이라면, `Isolation`은 여러 트랜잭션 사이의 경계입니다
- 격리 수준을 낮추면 동시성은 좋아질 수 있지만, `Dirty Read`, `Non-Repeatable Read`, `Phantom Read` 같은 이상 현상 가능성이 올라갑니다

격리 수준 자체는 이미 별도 글에서 자세히 다뤘으니, 여기서는 **ACID의 `I`가 바로 그 이야기**라고 연결해서 이해하면 됩니다.

> **참고:** 자세한 이상 현상과 격리 수준 차이는 트랜잭션 격리 수준 완전 정복, Dirty Read와 Phantom Read는 실제로 언제 발생할까, MVCC 완전 정복 글을 함께 보면 흐름이 이어집니다.

Phase 4. `Durability` — `COMMIT` 성공 뒤 장애가 나도 왜 결과가 남아야 할까요?

핵심: 커밋된 변경은 장애 후에도 살아남아야 합니다

PostgreSQL glossary는 `Durability`를 "**한 번 커밋된 트랜잭션의 변경이 시스템 장애 후에도 남아 있는 성질**"로 설명합니다.

예를 들어 이 상황을 생각해 보겠습니다.

```
START TRANSACTION;

UPDATE account
SET balance = balance - 10000
WHERE id = 1;

COMMIT;
```

애플리케이션은 `COMMIT` 성공 응답을 받았습니다. 그런데 바로 직후 DB 프로세스가 죽거나 서버 전원이 나갔다면, 방금 성공한 변경은 어떻게 되어야 할까요?

`Durability`가 보장된다면 **재시작 후에도 그 변경은 남아 있어야 합니다.**

MySQL InnoDB는 무엇에 의존할까요?

MySQL의 `InnoDB and the ACID Model` 문서는 `Durability`가 아래와 강하게 연결된다고 설명합니다.

- `innodb_flush_log_at_trx_commit`
- `sync_binlog` (`binary log`를 사용하는 환경)
- `doublewrite buffer`
- 저장장치의 write buffer
- 배터리 백업 캐시
- 운영체제의 `fsync()` 지원

즉, `Durability`는 단순히 SQL 문법만의 문제가 아니라 **로그 flush 전략, crash recovery, 운영체제, 스토리지 하드웨어까지 걸친 속성**입니다.

성능과 맞바꾸는 순간도 있습니다

MySQL 문서는 `Optimizing InnoDB Transaction Management`에서, 예상치 못한 종료 시 일부 최신 커밋 손실을 감수할 수 있다면 `innodb_flush_log_at_trx_commit=0`을 검토할 수 있다고 설명합니다.

이 말은 곧:

- 기본 설정에 가깝게 두면 더 강한 `Durability`
- flush를 느슨하게 하면 더 높은 처리량

사이의 트레이드오프가 있다는 뜻입니다.

그래서 실무에서는 이렇게 읽으면 됩니다

- `COMMIT` 성공은 "메모리에 잠깐 반영됨"이 아니라 **장애 후 복구 기준점이 생김**을 의미해야 합니다
- `Durability`는 DB 엔진만이 아니라 **스토리지와 운영체제 설정**에도 영향을 받습니다
- 성능 튜닝으로 flush 정책을 낮췄다면, **최신 커밋 일부 손실 가능성**을 팀이 명시적으로 알고 있어야 합니다

> **참고:** MySQL 문서도 `innodb_flush_log_at_trx_commit=1`이어도 운영체제나 하드웨어가 flush를 거짓 보고하면 완전한 보장을 약화시킬 수 있다고 설명합니다. 즉, `Durability`는 소프트웨어와 하드웨어가 함께 만드는 성질입니다.

Phase 5. ACID가 있어도 안 막아주는 것들은 무엇일까요?

1. 외부 시스템까지 자동으로 하나의 트랜잭션이 되지는 않습니다

DB 트랜잭션 안에서 주문 저장과 재고 차감은 묶을 수 있어도, 같은 흐름에서:

- 이메일 발송
- Kafka 발행
- Redis 갱신
- 다른 서비스 HTTP 호출

까지 자동으로 같은 로컬 트랜잭션으로 묶이진 않습니다.

즉, DB는 `COMMIT`됐는데 메일 발송은 실패할 수 있습니다. 이 문제는 `ACID`만으로 해결되지 않고, `Outbox Pattern`, 재시도, 보상 트랜잭션 같은 별도 설계가 필요합니다.

2. 잘못된 비즈니스 로직까지 고쳐주지는 않습니다

트랜잭션이 ACID를 만족해도:

- 잘못된 사용자에게 돈을 보냈거나
- 할인 계산식이 틀렸거나
- 만료 쿠폰을 발급했다면

그건 트랜잭션 속성 문제가 아니라 **업무 로직 문제**입니다.

3. 동시성 비용이 사라지지는 않습니다

격리를 높이면 안전해지지만:

- 락 대기
- 데드락
- 긴 트랜잭션으로 인한 purge 지연
- 처리량 감소

같은 비용이 따라옵니다.

즉, ACID는 공짜가 아닙니다. 특히 `Isolation`과 `Durability`는 성능과 자주 맞물립니다.

한눈에 보는 ACID 비교

ACID는 글자만 외우기보다, 각 속성이 막는 실패 모드를 구분해서 보는 편이 훨씬 실용적입니다.

속성	가장 짧은 의미	주로 막는 문제	MySQL InnoDB 기준 대표 수단
`Atomicity`	전부 성공하거나 전부 실패	출금만 되고 입금이 안 되는 부분 반영	트랜잭션, `COMMIT`, `ROLLBACK`, `autocommit`
`Consistency`	커밋 결과가 유효한 상태를 만족	깨진 참조, 제약 위반, 유효하지 않은 상태 커밋	제약 조건, 트랜잭션, crash-safe 복구
`Isolation`	동시 실행 간섭 제어	중간값 노출, 반복 조회 결과 뒤틀림, 팬텀	격리 수준, MVCC, consistent read, row lock
`Durability`	커밋 결과가 장애 후에도 유지	성공 응답 뒤 데이터 유실	redo log flush, crash recovery, `innodb_flush_log_at_trx_commit`, 스토리지 flush

실무에서는 이렇게 점검하면 됩니다

ACID를 읽고 나면 아래 질문으로 바로 연결해 보는 것이 좋습니다.

1. **이 작업은 정말 한 트랜잭션으로 묶여야 하나요?** 함께 실패해야 하는 SQL이라면 `Atomicity` 대상입니다.
2. **커밋 시점에 어떤 규칙이 반드시 유지돼야 하나요?** `UNIQUE`, `FOREIGN KEY`, `CHECK`, 애플리케이션 검증 중 어디에 둘지 결정해야 합니다.
3. **동시 요청이 들어오면 무엇을 보게 할 건가요?** 최신값, 스냅샷, 잠금 읽기 중 어떤 선택이 필요한지 봐야 합니다.
4. **`COMMIT` 성공 뒤 장애가 나면 어디까지 잃어도 되나요?** 이 질문이 `Durability` 설정과 인프라 선택을 바꿉니다.
5. **DB 밖의 부작용도 같은 성공/실패 단위로 묶어야 하나요?** 그렇다면 ACID만으로는 부족합니다.

정리

1. **`Atomicity`는 한 업무 단위를 중간 성공 없이 묶는 속성입니다.**
2. **`Consistency`는 커밋 시점의 결과가 무결성 규칙을 만족하는가를 묻습니다.** "내가 원하는 비즈니스 결과"와 완전히 같은 말은 아닙니다.
3. **`Isolation`은 동시에 실행되는 트랜잭션끼리 무엇을 보고 어디까지 간섭할 수 있는지 정하는 속성입니다.**
4. **`Durability`는 `COMMIT` 성공 후 장애가 나도 결과가 남아 있어야 한다는 속성입니다.**
5. **ACID는 데이터베이스 트랜잭션의 신뢰성을 설명하지만, 외부 시스템 연동이나 비즈니스 버그까지 자동으로 해결해 주지는 않습니다.**
6. **결국 ACID를 이해한다는 것은 `BEGIN` / `COMMIT` 문법을 아는 것이 아니라, 실패와 동시성 앞에서 무엇을 DB에 맡기고 무엇을 직접**

설계해야 하는지 구분할 수 있다는 뜻입니다.

Chapter 2

트랜잭션 격리 수준 완전 정복 — Read Uncommitted부터 Serializable까지

#동시성 제어 #트랜잭션

트랜잭션 격리 수준이 왜 필요한지, 각 레벨에서 어떤 이상 현상이 발생하는지 예제와 함께 정리합니다.

트랜잭션 격리, 왜 알아야 하나요?

두 사용자가 동시에 같은 데이터를 읽고 수정한다고 가정합니다. 아무 제어 없이 실행하면 어떤 일이 벌어질까요?

- 커밋되지 않은 데이터를 다른 트랜잭션이 읽어버립니다
- 같은 쿼리를 두 번 실행했는데 결과가 다릅니다
- 조건에 맞는 행 수가 쿼리할 때마다 바뀝니다

이런 문제들을 **이상 현상(Anomaly)** 이라 부르고, 트랜잭션 격리 수준은 이 이상 현상을 어디까지 허용할지 결정하는 설정입니다. 격리 수준이 높을수록 안전하지만 동시성이 떨어지고, 낮을수록 빠르지만 위험합니다.

이 글에서는 먼저 이상 현상 세 가지를 살펴보고, 네 가지 격리 수준이 각각 어떤 이상 현상을 방지하는지 예제와 함께 정리합니다.

Phase 1. 이상 현상 이해하기

격리 수준을 이해하려면 먼저 어떤 문제가 발생할 수 있는지 알아야 합니다. SQL 표준에서 정의하는 세 가지 이상 현상을 살펴보겠습니다.

Dirty Read — 커밋되지 않은 데이터를 읽는 문제

```
트랜잭션 A                              트랜잭션 B
──────────────────────────────────────────────────
UPDATE accounts
SET balance = 0
WHERE id = 1;
                                    SELECT balance
                                    FROM accounts
                                    WHERE id = 1;
                                    → 0 (커밋 안 된 값을 읽음)
ROLLBACK;
                                    → 0을 기반으로 로직 수행 (잘
못된 데이터!)
```

트랜잭션 A가 잔액을 0원으로 변경했지만 아직 커밋하지 않았습니다. 이 상태에서 트랜잭션 B가 0원을 읽고 로직을 수행합니다. 이후 A가 롤백하면 실제 잔액은 원래 값인데, B는 이미 0원을 기준으로 처리를 끝낸 상태입니다. 존재한 적 없는 데이터를 읽은 것이므로 **Dirty Read**라고 합니다.

Non-Repeatable Read — 같은 행을 두 번 읽었는데 값이 달라지는 문제

```
트랜잭션 A                              트랜잭션 B
──────────────────────────────────────────────────
SELECT balance
FROM accounts
WHERE id = 1;
→ 10000
                                        UPDATE accounts
                                        SET balance = 0
                                        WHERE id = 1;
                                        COMMIT;
SELECT balance
FROM accounts
WHERE id = 1;
→ 0 (같은 쿼리인데 결과가 다름!)
```

트랜잭션 A가 같은 `SELECT`를 두 번 실행했는데, 그 사이에 트랜잭션 B가 해당 행을 수정하고 커밋했습니다. 결과적으로 A는 같은 행을 읽었지만 값이 달라집니다. **반복 읽기가 보장되지 않는다**는 의미에서 Non-Repeatable Read라고 합니다.

Phantom Read — 같은 조건으로 조회했는데 행 수가 달라지는 문제

```
트랜잭션 A                              트랜잭션 B
──────────────────────────────────────────────────────
SELECT COUNT(*)
FROM orders
WHERE status = 'PAID';
→ 5
                                        INSERT INTO orders
(status)
                                        VALUES ('PAID');
                                        COMMIT;
SELECT COUNT(*)
FROM orders
WHERE status = 'PAID';
→ 6 (유령 행이 나타남!)
```

트랜잭션 A가 같은 조건으로 두 번 조회했는데, 그 사이에 트랜잭션 B가 조건에 맞는 새 행을 삽입하고 커밋했습니다. 갑자기 나타난 행을 **팬텀(Phantom)** 이라 부릅니다. Non-Repeatable Read가 **기존 행의 값 변경** 문제라면, Phantom Read는 **행 자체가 추가/삭제**되는 문제입니다.

> **참고:** Non-Repeatable Read와 Phantom Read의 차이를 혼동하기 쉽습니다. 핵심은 **대상**입니다. Non-Repeatable Read는 이미 읽은 **특정 행의 값**이 바뀌는 것이고, Phantom Read는 조건에 맞는 **행의 집합(결과 셋)**이 바뀌는 것입니다.

Phase 2. 네 가지 격리 수준

SQL 표준(SQL-92)은 네 가지 격리 수준을 정의합니다. 아래로 갈수록 격리가 강해집니다.

Read Uncommitted — 격리 없음

가장 낮은 격리 수준입니다. 다른 트랜잭션이 커밋하지 않은 변경 사항까지 읽을 수 있습니다.

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
```

- **Dirty Read** 발생 가능
- **Non-Repeatable Read** 발생 가능
- **Phantom Read** 발생 가능

사실상 격리가 없는 것과 같습니다. **실무에서 거의 사용하지 않습니다.** 굳이 쓴다면 정확한 데이터가 필요 없는 대략적인 통계 조회 정도에 한정됩니다.

Read Committed — 커밋된 데이터만 읽기

다른 트랜잭션이 **커밋한 데이터만** 읽을 수 있습니다.

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

- **Dirty Read** 방지
- **Non-Repeatable Read** 발생 가능
- **Phantom Read** 발생 가능

Dirty Read는 막아주지만, 같은 쿼리를 반복 실행하면 그 사이에 커밋된 변경 사항이 보일 수 있습니다.

```
트랜잭션 A                                트랜잭션 B
──────────────────────────────────────────────────────

SELECT balance FROM accounts
WHERE id = 1;
→ 10000
                                          UPDATE accounts
                                          SET balance = 5000
                                          WHERE id = 1;
                                          COMMIT;

SELECT balance FROM accounts
WHERE id = 1;
→ 5000 (커밋된 값이므로 읽기 허용)
```

트랜잭션 A 입장에서는 같은 쿼리를 두 번 실행했는데 결과가 달라졌지만, Read Committed에서는 이것이 정상 동작입니다.

Repeatable Read — 반복 읽기 보장

트랜잭션 내 **첫 번째 읽기 시점의 스냅샷을 기준으로** 데이터를 읽습니다. 같은 행에 대한 일반 `SELECT`를 여러 번 실행하면 같은 값을 반환합니다.

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

- **Dirty Read** 방지
- **Non-Repeatable Read** 방지
- **Phantom Read** SQL 표준에서는 발생 가능

MySQL(InnoDB)의 기본 격리 수준입니다. InnoDB는 MVCC(Multi-Version Concurrency Control)와 갭 락(Gap Lock)을 조합하여 Repeatable Read에서도 **대부분의 Phantom Read를 방지**합니다.

```
트랜잭션 A                              트랜잭션 B
──────────────────────────────────────────────────────
START TRANSACTION;
SELECT balance FROM accounts
WHERE id = 1;
→ 10000
                                    UPDATE accounts
                                    SET balance = 5000
                                    WHERE id = 1;
                                    COMMIT;
SELECT balance FROM accounts
WHERE id = 1;
→ 10000 (첫 번째 SELECT 시점의 스냅샷)
COMMIT;
```

트랜잭션 A는 첫 번째 SELECT 시점의 스냅샷을 계속 읽으므로, B가 중간에 값을 바꾸고 커밋해도 A에게는 보이지 않습니다.

> **참고:** MySQL InnoDB의 Repeatable Read는 SQL 표준보다 강력합니다. **일관된 읽기(Consistent Read)** 는 MVCC 스냅샷으로, **잠금 읽기(`SELECT ... FOR UPDATE`)와 쓰기**는 갭 락과 넥스트 키 락으로 Phantom을 줄입니다. 다만 일관된 읽기와 잠금 읽기/쓰기를 한 트랜잭션 안에서 섞으면, 고전적인 Phantom Read와는 조금 다른 형태의 일관성 혼란이 생길 수 있습니다.

Serializable — 완전한 격리

가장 높은 격리 수준입니다. 트랜잭션들이 마치 **하나씩 순서대로 실행되는 것처럼** 동작합니다.

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

- **Dirty Read** 방지
- **Non-Repeatable Read** 방지
- **Phantom Read** 방지

모든 이상 현상을 차단하지만, 그만큼 **동시성이 크게 떨어집니다.** MySQL InnoDB에서는 `autocommit`이 꺼진 `SERIALIZABLE` 트랜잭션에서 일반 `SELECT`도 사실상 `SELECT ... FOR SHARE`처럼 동작하여 읽기에도 공유 락이 걸립니다.

```
트랜잭션 A                              트랜잭션 B
──────────────────────────────────────────────────────
SELECT COUNT(*)
FROM orders
WHERE status = 'PAID';
→ 5 (공유 락 설정)
                                        INSERT INTO orders
(status)
                                        VALUES ('PAID');
                                        → 대기 (A의 락이 해제될 때까
지 블로킹)
COMMIT;
                                        → INSERT 실행
```

트랜잭션 A가 읽은 범위에 락이 걸리므로, B는 A가 커밋할 때까지 해당 범위에 쓰기를 할 수 없습니다. 안전하지만 **데드락 발생 확률이 높아지고 처리량이 급격히 감소**합니다.

Phase 3. 격리 수준의 구현 — 락과 MVCC

격리 수준을 공부하다 보면 락(Lock)과 헷갈리기 쉽습니다. 이 둘은 **레이어가 다릅니다.**

- **격리 수준** — 다른 트랜잭션의 변경이 **보이느냐 안 보이느냐**를 결정하는 **정책**입니다
- **락** — 다른 트랜잭션이 해당 데이터에 **접근 자체를 못하게 차단**하는 **메커니즘**입니다

상품 재고가 1개 남은 상황에서 두 사용자가 동시에 주문하는 경우로 비교하면 명확합니다.

```
-- 격리 수준이 하는 일: "뭘 보여줄지" 결정
-- A가 재고를 읽은 뒤, B가 재고를 0으로 변경하고 커밋
-- → Read Committed: A의 다음 SELECT에서 0이 보임
-- → Repeatable Read: A의 다음 SELECT에서 여전히 1이 보임
-- → 어느 쪽이든 B의 UPDATE 자체를 막지는 않음

-- 락이 하는 일: "접근 자체를 차단"
SELECT stock FROM products WHERE id = 1 FOR UPDATE;  -- A가 배타 락 획득
-- → B의 잠금 읽기(FOR UPDATE, FOR SHARE)와 쓰기는 A가 커밋할 때까지 대기
-- → 단, 일반 SELECT(스냅샷 읽기)는 락과 무관하게 동작
```

격리 수준은 **"창문 유리"** 로, 밖이 보이느냐 안 보이느냐를 결정합니다. 락은 **"문 잠금"** 으로, 아예 들어오지 못하게 막습니다. DB는 내부적으로 **락과 MVCC를 조합**하여 각 격리 수준의 정책을 구현합니다.

락 기반 (Lock-Based)

가장 직관적인 방법입니다. 데이터를 읽거나 쓸 때 **락(Lock)을 획득**하고, 다른 트랜잭션이 접근하지 못하게 막습니다.

락 종류	설명
공유 락 (S)	읽기 락. 여러 트랜잭션이 동시에 획득 가능
배타 락 (X)	쓰기 락. 하나의 트랜잭션만 획득 가능
갭 락 (Gap)	인덱스 레코드 사이의 간격을 잠금
넥스트 키 락	레코드 락 + 갭 락의 조합

격리 수준이 높아질수록 락의 범위가 넓어지고 보유 시간이 길어집니다.

MVCC (Multi-Version Concurrency Control)

락 없이 읽기 일관성을 제공하는 방법입니다. 데이터를 변경하면 이전 버전을 별도로 보관하고, 각 트랜잭션은 격리 수준에 따라 자신에게 보여야 할 버전을 읽습니다.

```
행 id=1의 버전 히스토리 (트랜잭션 A는 txn_100 시점에 시작):

[balance=10000, modified_by=txn_50]   ← 이전 버전 (Undo 로그에 보관)
          ↓
[balance=5000, modified_by=txn_105]   ← 최신 버전 (데이터 페이지)
```

MySQL InnoDB에서는 **Undo 로그**에 이전 버전을 보관합니다. 예를 들어 Repeatable Read에서 트랜잭션 A(txn_100)는 자신의 스냅샷 이후에 커밋된 txn_105의 변경을 무시하고, txn_50이 남긴 이전 버전

(balance=10000)을 읽습니다. 반면 Read Committed에서는 다음 쿼리 시점에 txn_105가 이미 커밋되었다면 최신 버전(balance=5000)이 보입니다.

- **Read Committed** — 매 쿼리마다 새로운 스냅샷을 생성합니다
- **Repeatable Read** — 트랜잭션 내 첫 번째 읽기 시점에 스냅샷을 생성하고 끝까지 유지합니다

이 차이가 Non-Repeatable Read 발생 여부를 결정합니다.

> **참고:** MVCC 덕분에 읽기와 쓰기가 서로를 블로킹하지 않습니다. Repeatable Read에서도 `SELECT`에 락을 걸지 않고 스냅샷을 읽으므로 높은 동시성을 유지할 수 있습니다. 이것이 MySQL이 Repeatable Read를 기본값으로 선택할 수 있는 이유입니다.

Phase 4. 실무에서 주의할 점

현재 격리 수준 확인하기

```
SELECT @@transaction_isolation;
```

사용 중인 DB의 기본 격리 수준을 반드시 파악하고 있어야 합니다. MySQL은 `REPEATABLE-READ`가 기본값입니다.

Lost Update 문제

읽기-계산-쓰기 패턴에서 발생하는 대표적인 동시성 문제입니다.

```
트랜잭션 A                              트랜잭션 B
──────────────────────────────────────────────────────
SELECT balance FROM accounts
WHERE id = 1;
→ 10000
                                        SELECT balance FROM
accounts
                                        WHERE id = 1;
                                        → 10000
UPDATE accounts
SET balance = 10000 - 3000
WHERE id = 1;
COMMIT;
                                        UPDATE accounts
                                        SET balance = 10000 -
5000
                                        WHERE id = 1;
                                        COMMIT;
→ 최종 잔액: 5000 (A의 출금 3000이 사라짐!)
```

두 트랜잭션이 같은 값을 읽고 각자 계산한 결과를 상수로 덮어쓰면서 A의 변경이 유실됩니다. **MySQL InnoDB의 Repeatable Read는 이런 read-modify-write 패턴의 Lost Update를 자동으로 감지하지 못할 수 있습니다.**

해결: 비관적 락 또는 원자적 연산

```
-- 방법 1: SELECT ... FOR UPDATE (비관적 락)
SELECT balance FROM accounts WHERE id = 1 FOR UPDATE;
-- 이 행에 배타 락이 걸려 다른 트랜잭션의 잠금 읽기와 쓰기가 대기

-- 방법 2: 원자적 UPDATE (읽기 없이 한 번에 처리)
UPDATE accounts SET balance = balance - 3000 WHERE id = 1;
```

`SELECT ... FOR UPDATE`는 해당 행에 배타 락을 걸어 다른 트랜잭션의 **잠금 읽기(`FOR UPDATE`, `FOR SHARE`)와 쓰기를 블로킹**합니다. 단, 일반 `SELECT`(MVCC 스냅샷 읽기)는 락과 무관하게 동작하므로 차단되지 않습니다. 원자적 `UPDATE`는 DB가 최신 값을 기준으로 한 문장에서 계산하므로, 예시처럼 애플리케이션이 읽은 값을 상수로 다시 덮어쓰는 패턴보다 안전합니다.

데드락에 대비하기

격리 수준이 높을수록 락 범위가 넓어지고 데드락 발생 확률이 올라갑니다.

```
트랜잭션 A                              트랜잭션 B
──────────────────────────────────────────────────────

UPDATE accounts
SET balance = balance - 1000
WHERE id = 1;  (행 1에 X락)
                                        UPDATE accounts
                                        SET balance = balance
- 1000
                                        WHERE id = 2;  (행 2에 X
락)
UPDATE accounts
SET balance = balance + 1000
WHERE id = 2;  → 대기 (B가 행 2 보유)
                                        UPDATE accounts
                                        SET balance = balance
+ 1000
                                        WHERE id = 1;  → 대기 (A
가 행 1 보유)
→ 데드락!
```

실무에서의 대응 방법입니다.

- **락 획득 순서를 통일** — 항상 `id` 오름차순으로 락을 획득하면 순환 대기가 발생하지 않습니다
- **트랜잭션을 짧게 유지** — 락 보유 시간이 줄어들어 충돌 확률이 낮아집니다
- **데드락 감지에 의존** — MySQL InnoDB는 데드락을 자동 감지하고 한쪽 트랜잭션을 롤백합니다. 애플리케이션에서 재시도 로직을 구현합니다

> **참고:** Spring에서 `@Transactional`을 사용할 때 격리 수준을 지정할 수 있습니다. `@Transactional(isolation = Isolation.REPEATABLE_READ)`처럼 선언하면 해당 메서드의 트랜잭션에만 격리 수준이 적용됩니다.

한눈에 보는 격리 수준 비교

격리 수준	Dirty Read	Non-Repeatable Read	Phantom Read	성능
Read Uncommitted	발생	발생	발생	가장 빠름
Read Committed	방지	발생	발생	빠름
Repeatable Read	방지	방지	발생 가능*	보통
Serializable	방지	방지	방지	느림

* MySQL InnoDB는 일관된 읽기와 잠금 읽기/쓰기를 다르게 처리하므로, 표준 Repeatable Read보다 더 강하게 동작하는 경우가 많습니다.

정리

1. **Dirty Read** — 커밋되지 않은 데이터를 읽는 문제입니다. Read Committed 이상이면 방지됩니다

2. **Non-Repeatable Read** — 같은 행을 두 번 읽었을 때 값이 달라지는 문제입니다. Repeatable Read 이상이면 방지됩니다
3. **Phantom Read** — 같은 조건 조회 시 행의 수가 달라지는 문제입니다. Serializable에서 완전히 방지됩니다
4. **MVCC** — 락 없이 읽기 일관성을 제공하는 핵심 메커니즘입니다. 격리 수준에 따라 스냅샷 생성 시점이 달라집니다
5. **Lost Update** — 특히 애플리케이션이 읽은 값을 계산한 뒤 상수로 덮어쓰는 read-modify-write 패턴에서 문제 됩니다. MySQL InnoDB RR은 자동 감지하지 못할 수 있으므로 `SELECT ... FOR UPDATE` 또는 원자적 연산으로 방지합니다
6. **실무 판단** — 많은 서비스는 DB 기본값인 Repeatable Read로도 시작할 수 있으며, 특별한 요구사항이 있는 트랜잭션에만 격리 수준을 개별 조정합니다

Chapter 3

Dirty Read와 Phantom Read는 실제로 언제 발생할까 — 교과서와 실무 사이의 간격

#동시성 제어 #트랜잭션

`Dirty Read`가 왜 실무에서 거의 안 보이는지, `Phantom Read`가 왜 MySQL InnoDB에서는 다른 문제처럼 관찰되는지 실제 발생 조건 중심으로 정리합니다.

Dirty Read와 Phantom Read, 왜 헷갈릴까요?

트랜잭션 격리 수준, MVCC, REPEATABLE READ 실전 사례 글까지 읽고 나면 이런 질문이 남습니다.

- `Dirty Read`는 교과서에서 꼭 나오는데, 왜 실무에서는 거의 못 볼까요?
- `Phantom Read`는 분명 배웠는데, MySQL InnoDB에서는 왜 "행이 갑자기 늘어났다"보다 다른 형태로 체감될까요?
- 내가 겪은 문제는 정말 `Dirty Read`나 `Phantom Read`였을까요, 아니면 다른 동시성 문제였을까요?

핵심은 **이상 현상 이름과 실무에서 관찰되는 증상이 꼭 같은 모습으로 나타나지 않는다**는 점입니다.

- `Dirty Read`는 **발생 조건 자체가 꽤 제한적**입니다
- `Phantom Read`는 **같은 트랜잭션 안에서 같은 범위 조건을 반복 조회**해야 의미가 있습니다
- MySQL InnoDB의 기본 격리 수준인 `REPEATABLE READ`에서는, 고전적인 `Phantom Read`보다 **스냅샷 읽기, 범위 락 대기, 데드락**처럼 보이는 경우가 더 많습니다

이 글은 MySQL InnoDB를 기준으로, `Dirty Read`와 `Phantom Read`가 **실제로 언제 발생하고**, 왜 **실무에서는 다른 문제처럼 보이는지**를 정리합니다.

먼저 가장 짧은 답부터 보면

현상	실제 발생 조건	실무에서 잘 안 보이거나 다르게 보이는 이유
`Dirty Read`	다른 트랜잭션이 아직 커밋하지 않은 값을 읽음	보통 `READ UNCOMMITTED`가 필요하고, 운영 기본값은 대개 그보다 높습니다
`Phantom Read`	같은 트랜잭션이 같은 조건 범위 조회를 두 번 했는데, 중간에 다른 트랜잭션이 조건에 맞는 행을 추가/삭제/변경해 결과 집합이 바뀜	MySQL InnoDB의 `REPEATABLE READ`에서는 일반 `SELECT`는 스냅샷을 보고, 범위 잠금 읽기는 새 행 진입을 막는 쪽으로 동작합니다
실무에서 더 자주 보는 모습	`stale snapshot`, 범위 락 대기, 데드락, 중복 부작용	이상 현상 이름보다 DB 엔진의 구현 방식이 운영 증상에 더 직접적으로 드러납니다

가장 짧게 줄이면 이렇습니다.

- **`Dirty Read`는 낮은 격리 수준을 일부러 열어둬야 잘 발생합니다**
- **`Phantom Read`는 같은 트랜잭션 안에서 범위 조회를 반복해야 의미가 있습니다**
- **MySQL InnoDB에서는 팬텀보다 "왜 나는 예전 결과를 계속 보지?" 또는 "왜 저 `INSERT`가 대기하지?"로 만나는 경우가 더 많습니다**

Phase 1. `Dirty Read`는 정확히 언제 발생할까?

문제: "값이 달라졌다"와 `Dirty Read`를 같은 말로 쓰기 쉽다

실무에서 이런 표현을 자주 봅니다.

- "방금 다른 요청이 값을 바꿔서 결과가 달라졌어요"
- "캐시와 DB가 잠깐 달라 보여요"
- "`read replica`에서 예전 값이 보여요"

하지만 이 셋은 `Dirty Read`가 아닐 수 있습니다. `Dirty Read`는 정의가 더 엄격합니다.

> **다른 트랜잭션이 아직 커밋하지 않은 값을 읽었고, 그 값이 나중에 롤백될 수도 있는 상태**

즉, **세상에 최종적으로 존재하지 않을 수도 있는 값을 읽어야** `Dirty Read`입니다.

예시: 커밋 전 잔액을 읽어버리는 경우

```
트랜잭션 A                              트랜잭션 B
──────────────────────────────────────────────────
UPDATE accounts
SET balance = 0
WHERE id = 1;
                                    SELECT balance
                                    FROM accounts
                                    WHERE id = 1;
                                    → 0
ROLLBACK;
                                    → 실제로는 존재하지 않을 값
기준으로 처리
```

여기서 `트랜잭션 B`가 읽은 `0`은 커밋된 적이 없습니다. 이것이 `Dirty Read` 입니다.

실무에서 언제 진짜로 보일까?

실무에서 `Dirty Read`가 보이려면 보통 아래 조건이 필요합니다.

1. **격리 수준이 `READ UNCOMMITTED` 여야 합니다**
2. 또는 그와 비슷하게 **커밋 전 데이터를 읽게 만드는 설정/힌트**가 있어야 합니다
3. 그리고 읽은 쪽이 **정말로 커밋 전 값을 봐야** 합니다

MySQL InnoDB 기본값은 `REPEATABLE READ`입니다. 그래서 별도 설정 없이 운영하면 `Dirty Read`는 거의 나오지 않습니다.

실제로는 이런 경우에 가끔 후보가 됩니다.

- 정확도보다 대기 회피를 우선해서 **낮은 격리 수준으로 돌린 운영성 조회**
- 통계성 배치에서 **대략적인 값만 보겠다**고 `READ UNCOMMITTED`를 쓴 경우
- 팀이 의도를 잘 모른 채 **세션 격리 수준을 낮춘 뒤 그대로 유지**한 경우

반대로 아래는 `Dirty Read`가 아닙니다.

- 다른 요청이 먼저 커밋해서 값이 달라진 것
- 읽기 복제본 지연 때문에 예전 값이 보이는 것
- 캐시가 늦게 갱신되어 DB와 잠시 어긋나는 것

판단 기준은 간단합니다.

- **내가 읽은 값이 커밋 전 값이었는가**
- **그 값이 나중에 롤백될 수 있었는가**

이 둘이 아니면 보통 `Dirty Read`가 아닙니다.

Phase 2. `Phantom Read`는 정확히 언제 발생할까?

문제: 결과가 달라졌다고 다 `Phantom Read`는 아니다

`Phantom Read`도 실무에서 자주 과하게 넓게 쓰입니다.

- 첫 번째 API 요청에서는 목록이 10건이었는데, 두 번째 요청에서는 11건입니다
- 페이지 1을 보고 페이지 2로 갔더니 중간에 새 글이 끼어들었습니다
- 다른 사용자가 주문을 추가해서 카운트가 달라졌습니다

이 중 많은 경우는 **고전적인 `Phantom Read`가 아닙니다.** 왜냐하면 `Phantom Read`는 보통 **같은 트랜잭션 안에서 같은 조건 조회를 반복**해야 하기 때문입니다.

정의는 이렇습니다.

> **같은 트랜잭션이 같은 조건으로 두 번 조회했는데, 그 사이에 다른 트랜잭션이 조건에 맞는 행을 추가/삭제/변경해서 결과 집합이 달라지는 현상**

즉, 핵심은 **같은 트랜잭션**, **같은 조건**, **행 집합 변화**입니다.

예시: `READ COMMITTED`에서 범위 결과가 늘어나는 경우

```
-- 세션 A
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
START TRANSACTION;

SELECT COUNT(*)
FROM coupons
WHERE user_id = 1
  AND status = 'ACTIVE';

-- 0
```

```
-- 세션 B
INSERT INTO coupons(user_id, status)
VALUES (1, 'ACTIVE');

COMMIT;
```

```
-- 세션 A
SELECT COUNT(*)
FROM coupons
WHERE user_id = 1
  AND status = 'ACTIVE';

-- 1
COMMIT;
```

세션 A 입장에서는 같은 조건으로 두 번 조회했는데 결과 집합이 바뀌었습니다. 이것이 전형적인 `Phantom Read` 입니다.

실무에서 특히 의미가 커지는 순간

`Phantom Read` 는 **범위 조건을 믿고 비즈니스 판단을 할 때** 문제가 됩니다. 예를 들면:

- "현재 활성 쿠폰이 0건이면 발급 가능" 같은 **존재 여부 판단**
- "`status = 'WAITING'` 인 작업 개수를 보고 배치 크기를 정하는" **범위 카운트 기반 판단**
- "`scheduled_at ⩽ NOW()` 조건을 만족하는 작업만 집계" 같은 **시점 기반 스캔**

즉, 단순히 목록이 달라졌다는 것보다, **그 달라진 결과를 기준으로 로직을 이어가는 순간** 문제가 됩니다.

Phase 3. 그런데 왜 MySQL InnoDB에서는 팬텀이 잘 안 보일까?

문제: 교과서대로 재현하려고 하면 MySQL에서는 다른 결과가 나온다

트랜잭션 격리 수준 글에서 봤듯이, SQL 표준 관점에서는 `REPEATABLE READ` 에서도 `Phantom Read` 가 발생할 수 있습니다.

하지만 MySQL InnoDB에서는 실제로 체감이 다릅니다. 이유는 두 갈래입니다.

- 일반 `SELECT`는 **MVCC 스냅샷 읽기**를 합니다
- 범위를 보호하는 잠금 읽기와 쓰기에서는 **갭 락 / 넥스트 키 락**이 개입할 수 있습니다

즉, "행이 갑자기 나타났다"보다 아래 두 형태로 더 자주 보입니다.

- **같은 트랜잭션인데도 나는 예전 범위 결과를 계속 본다**
- **새 행을 넣으려는 쪽이 대기하거나 데드락 난다**

경우 1. 일반 `SELECT`에서는 팬텀 대신 스냅샷이 유지된다

```
-- 세션 A
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
START TRANSACTION;

SELECT COUNT(*)
FROM coupons
WHERE user_id = 1
  AND status = 'ACTIVE';

-- 0
```

```
-- 세션 B
INSERT INTO coupons(user_id, status)
VALUES (1, 'ACTIVE');

COMMIT;
```

```
-- 세션 A
SELECT COUNT(*)
FROM coupons
WHERE user_id = 1
  AND status = 'ACTIVE';

-- 여전히 0
COMMIT;
```

이 경우 세션 A는 같은 트랜잭션 안에서 **첫 스냅샷 기준의 결과**를 계속 봅니다. 그래서 고전적인 `Phantom Read`는 잘 드러나지 않습니다.

이 현상은 많은 개발자가 처음엔 버그처럼 느끼는 부분입니다. 하지만 InnoDB 기준으로는 **정상적인 스냅샷 읽기 동작**입니다.

경우 2. 잠금 읽기에서는 새 행이 못 들어오게 막는다

이번에는 잠금 읽기로 범위를 읽는다고 가정해 보겠습니다.

```
-- 세션 A
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
START TRANSACTION;

SELECT id
FROM coupons
WHERE user_id = 1
  AND status = 'ACTIVE'
FOR UPDATE;
```

이후 세션 B가 같은 범위에 해당하는 행을 넣으려 하면, InnoDB는 **적절한 인덱스를 타는 범위 잠금 읽기**라는 전제에서 인덱스 탐색과 락 범위에 따라 **대기**시키거나 상황에 따라 **데드락**으로 정리할 수 있습니다.

```
-- 세션 B
INSERT INTO coupons(user_id, status)
VALUES (1, 'ACTIVE');

-- 세션 A가 끝날 때까지 대기할 수 있음
```

즉, 교과서에서 "팬텀이 나타난다"로 설명되던 경쟁이, MySQL InnoDB에서는 실무상 이렇게 보일 수 있습니다.

- `INSERT`가 갑자기 오래 대기한다
- 범위 조회 뒤 쓰기에서 데드락이 발생한다
- 팬텀이 보이기보다, 아예 그 범위로 새 행이 못 들어간다

그래서 InnoDB에서는 `Phantom Read`를 **결과가 달라지는 현상**으로만 기억하면 운영 증상을 놓치기 쉽습니다. **범위 경쟁이 락으로 흡수된 결과**까지 같이 봐야 합니다.

Phase 4. 실무에서는 무엇으로 관찰될까?

`Dirty Read`와 `Phantom Read`를 교과서 이름 그대로 만나는 경우보다, 아래 증상으로 만나는 경우가 더 많습니다.

관찰된 증상	실제로 먼저 의심할 것
어떤 값이 잠깐 보였다가 나중에 사라짐	정말 `READ UNCOMMITTED` 였는지, 읽은 값이 커밋 전 값이었는지
같은 트랜잭션에서 범위 카운트가 바뀜	`READ COMMITTED` 인지, 범위 조건을 두 번 읽었는지
같은 트랜잭션인데도 최신 삽입이 안 보임	`REPEATABLE READ` 의 스냅샷 읽기인지
범위 조건 뒤 `INSERT` 가 대기함	갭 락 / 넥스트 키 락 같은 범위 락 경쟁인지
읽기 결과를 믿고 부작용을 실행했더니 뒤에서 꼬임	스냅샷 `stale decision` 인지, 불변식을 읽기 대신 쓰기에서 강제해야 하는지

특히 마지막 줄은 실무에서 매우 중요합니다.

REPEATABLE READ 실전 사례 글에서 본 것처럼, 실제 장애는 꼭 "`Phantom Read` 가 났다"의 모습으로 오지 않습니다. 오히려:

- 읽을 때는 과거 스냅샷을 보고
- 애플리케이션은 그것을 최신 상태라고 믿고
- 쓰기는 최신 행에 적용되면서
- 부가 이력이나 외부 부작용이 중복되는

형태로 더 자주 나타납니다.

즉, 실무에서는 이상 현상 이름 자체보다 **읽기 기준과 쓰기 기준이 어긋났는가**를 먼저 보는 편이 더 유용합니다.

Phase 5. 그래서 어떻게 판단하고 대응할까?

1. `Dirty Read`가 의심되면 먼저 격리 수준부터 확인합니다

`Dirty Read`는 보통 엔진 기본 동작이 아니라 **설정이 열려 있어야** 발생합니다.

- 세션 격리 수준이 `READ UNCOMMITTED`인지
- 특정 조회만 낮은 격리 수준으로 돌리고 있는지
- 운영성/통계성 쿼리가 정확도를 포기한 설정인지

이걸 먼저 확인하는 편이 맞습니다.

2. `Phantom Read`가 의심되면 "같은 트랜잭션, 같은 조건"인지부터 봅니다

많은 경우는 사실 `Phantom Read`가 아니라:

- 서로 다른 HTTP 요청 사이의 자연스러운 결과 변화
- `OFFSET` 페이지네이션의 구조적 흔들림
- `REPEATABLE READ`의 스냅샷 읽기

인 경우가 더 많습니다.

즉, 먼저 이 질문을 해야 합니다.

- 정말 **같은 트랜잭션** 안에서 벌어진 일인가?

- 정말 **같은 조건 범위 조회**를 반복했는가?
- 그 결과 집합 변화가 **비즈니스 판단에 직접 사용되었는가?**

3. 해결은 격리 수준 하나로 끝나지 않는 경우가 많습니다

범위 판단을 더 안전하게 만들고 싶다면, 실무에서는 보통 아래 선택지를 같이 봅니다.

- 조건부 `UPDATE`
- `UNIQUE` 제약 조건
- `SELECT ... FOR UPDATE` 같은 잠금 읽기
- 작업 중복이라면 `named lock` 또는 `distributed lock`

즉, 문제를 "`Phantom Read`니까 무조건 `SERIALIZABLE`"로 가는 것이 아니라, **무엇을 불변식으로 지켜야 하는가**로 다시 번역해야 합니다.

분산 락은 언제 써야 할까, 멱등성 완전 정복 글과도 연결되는 지점입니다.

정리

1. **`Dirty Read`는 커밋 전 값을 읽어야만 성립합니다** — 그래서 기본 운영 설정에서는 의외로 보기 어렵습니다
2. **`Phantom Read`는 같은 트랜잭션 안의 같은 범위 조회가 다시 달라질 때 성립합니다** — 단순히 요청 간 결과가 달라지는 것과는 다릅니다
3. **MySQL InnoDB의 `REPEATABLE READ`에서는 고전적인 팬텀보다 스냅샷 읽기와 범위 락 경쟁으로 더 자주 체감됩니다**

4. **실무에서는 이상 현상 이름보다 관찰 증상을 먼저 해석해야 합니다** — `stale snapshot`인지, 범위 락 대기인지, 정말 낮은 격리 수준 문제인지 구분해야 합니다
5. **대응은 격리 수준 조정보다 불변식을 어디서 강제할지에 더 가까운 경우가 많습니다** — 조건부 `UPDATE`, `UNIQUE`, 잠금 읽기, 멱등성, 분산 락까지 함께 검토해야 합니다

PART 2

2부. 동시성 제어 메커니즘

Chapter 4

MVCC 완전 정복 — Undo 로그부터 스냅샷 읽기까지

#동시성 제어

MVCC가 왜 필요한지, Undo 로그와 스냅샷 읽기가 어떻게 동작하는지 정리합니다.

MVCC, 왜 알아야 하나요?

트랜잭션 격리 수준 글과 데이터베이스 락 글을 읽다 보면 이런 궁금증이 생깁니다.

- `Repeatable Read`인데 왜 일반 `SELECT`는 막히지 않을까요?
- 다른 트랜잭션이 값을 바꿨는데, 나는 왜 예전 값을 계속 읽을 수 있을까요?
- 락을 줄이면서도 읽기 일관성은 어떻게 유지할까요?

이 질문의 핵심에 있는 것이 **MVCC(Multi-Version Concurrency Control)** 입니다. MVCC는 데이터를 한 버전만 두지 않고 **여러 버전으로 관리**해서, 읽기와 쓰기가 서로를 덜 막으면서도 일관성을 유지하게 해줍니다.

이 글에서는 먼저 MVCC가 필요한 이유를 살펴보고, InnoDB 기준으로 **Undo 로그, Read View, 스냅샷 읽기**가 어떻게 동작하는지 정리합니다.

Phase 1. MVCC가 필요한 이유

락만으로 동시성을 제어하면 가장 단순합니다. 읽을 때는 공유 락, 쓸 때는 배타 락을 걸면 됩니다. 문제는 **읽기까지 서로 막히기 시작한다**는 점입니다.

락만으로 읽기를 보호하면 어떤 일이 생길까?

```
트랜잭션 A                              트랜잭션 B
──────────────────────────────────────────────────
SELECT balance
FROM accounts
WHERE id = 1;  (공유 락)
                                        UPDATE accounts
                                        SET balance = 5000
                                        WHERE id = 1;
                                        → 대기 (A의 공유 락 해제까
지)
SELECT balance
FROM accounts
WHERE id = 1;  (다시 읽기)
COMMIT;
                                        → UPDATE 실행
```

이 방식은 읽기 일관성은 확보할 수 있지만, **읽기 때문에 쓰기가 막히고**, 반대로 **쓰기 때문에 읽기가 막히는 상황**이 자주 생깁니다. 동시성이 크게 떨어집니다.

MVCC의 핵심 아이디어

MVCC는 데이터를 수정할 때 기존 값을 바로 덮어쓴 뒤 끝내지 않고, **이전 버전을 별도로 보관**합니다. 그래서 읽는 쪽은 자신이 봐야 할 시점의 버전을 읽고, 쓰는 쪽은 최신 버전을 갱신할 수 있습니다.

```
최신 버전: balance = 5000
이전 버전: balance = 10000

읽는 트랜잭션 A → 자신의 스냅샷에 맞는 버전 선택
쓰는 트랜잭션 B → 최신 버전 수정
```

즉, MVCC는 **"누가 먼저 락을 잡았는가"** 보다 **"이 트랜잭션에게 어떤 버전을 보여줘야 하는가"** 에 집중하는 방식입니다.

> **참고:** MVCC가 락을 완전히 없애는 것은 아닙니다. 일반 `SELECT` 같은 **스냅샷 읽기**를 락 없이 처리할 수 있게 해주지만, `UPDATE`, `DELETE`, `SELECT ... FOR UPDATE` 같은 쓰기나 잠금 읽기에는 여전히 락이 필요합니다.

Phase 2. InnoDB에서 MVCC는 어떻게 동작할까?

InnoDB 기준으로 보면 MVCC는 크게 세 가지 요소로 설명할 수 있습니다.

1. **최신 레코드** — 현재 데이터 페이지에 있는 최신 값
2. **Undo 로그** — 이전 버전으로 되돌릴 수 있는 정보

3. **Read View** — 지금 이 트랜잭션이 어떤 버전을 볼 수 있는지 판단하는 기준

최신 레코드와 Undo 로그

예를 들어 계좌 잔액이 10000원인 행이 있다고 가정합니다.

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

이 `UPDATE`가 실행되면 InnoDB는 최신 레코드를 5000으로 바꾸고, 이전 값 10000은 **Undo 로그**에 기록합니다.

```
데이터 페이지(최신):
[id=1, balance=5000]

Undo 로그(이전 버전):
[id=1, balance=10000]
```

이전 버전이 따로 남아 있으므로, 어떤 트랜잭션은 최신 값 5000을 읽고, 다른 트랜잭션은 예전 값 10000을 읽을 수 있습니다.

버전 체인

행이 여러 번 수정되면 Undo 로그는 하나만 생기지 않습니다. 수정될 때마다 **이전 버전이 체인처럼 연결**됩니다.

```
최신 레코드: balance = 3000
    ↓
Undo #1: balance = 5000
    ↓
Undo #2: balance = 10000
```

트랜잭션은 자신의 스냅샷에 맞는 버전을 찾을 때까지 이 체인을 따라 내려갑니다.

숨은 컬럼

InnoDB의 각 행에는 내부적으로 MVCC 판단에 필요한 정보가 함께 붙어 있습니다.

- `DB_TRX_ID` — 이 행을 마지막으로 수정한 트랜잭션 ID
- `DB_ROLL_PTR` — 이전 버전(Undo 로그)으로 가는 포인터

우리가 직접 보는 컬럼은 아니지만, InnoDB는 이 정보를 이용해 **이 버전이 내 스냅샷에서 보여야 하는지** 판단합니다.

Phase 3. Read View — 어떤 버전을 보여줄지 결정하는 기준

Undo 로그만 있다고 해서 MVCC가 완성되지는 않습니다. 여러 버전 중에서 **지금 이 트랜잭션이 어떤 버전을 읽어야 하는지** 결정하는 기준이 필요합니다. InnoDB에서는 이것을 **Read View**라고 부릅니다.

Read View를 쉽게 이해하기

트랜잭션 A가 시작된 시점에 활성 트랜잭션이 다음과 같다고 가정합니다.

```
활성 트랜잭션: 101, 102, 105
새로 시작한 트랜잭션 A의 시점
```

이때 트랜잭션 A는 대략 이런 기준을 갖습니다.

- **이미 오래전에 커밋된 트랜잭션**이 만든 버전은 볼 수 있습니다
- **아직 커밋되지 않은 트랜잭션**이 만든 버전은 볼 수 없습니다
- **내가 직접 수정한 내용**은 볼 수 있습니다

즉, Read View는 "지금 시점에서 커밋이 끝난 버전만 본다"는 규칙을 트랜잭션 단위로 들고 있는 셈입니다.

버전 선택 예시

```
행 id=1의 버전 히스토리

[balance=3000, modified_by=txn_110]   ← 최신 버전
        ↓
[balance=5000, modified_by=txn_105]
        ↓
[balance=10000, modified_by=txn_90]
```

트랜잭션 A의 Read View에서 `txn_105`, `txn_110`이 아직 보이면 안 되는 상태라면, A는 위에서부터 내려가다가 **처음으로 읽을 수 있는 버전인 `txn_90` 의 값 10000**을 읽습니다.

이것이 "다른 트랜잭션이 값을 바꿨는데도, 나는 예전 값을 계속 읽는다"는 현상의 정체입니다.

Read Committed와 Repeatable Read의 차이

같은 MVCC라도 **언제 Read View를 새로 만들 것인가**에 따라 격리 수준의 동작이 달라집니다.

격리 수준	Read View 생성 시점	결과
Read Committed	`SELECT`마다 새로 생성	같은 트랜잭션에서도 다시 읽으면 값이 달라질 수 있음
Repeatable Read	트랜잭션의 첫 일관된 읽기 시점에 생성	같은 트랜잭션에서는 같은 값을 계속 읽음

```
-- Read Committed
SELECT balance FROM accounts WHERE id = 1;   -- 10000
-- 다른 트랜잭션이 5000으로 수정 후 COMMIT
SELECT balance FROM accounts WHERE id = 1;   -- 5000

-- Repeatable Read
SELECT balance FROM accounts WHERE id = 1;   -- 10000
-- 다른 트랜잭션이 5000으로 수정 후 COMMIT
SELECT balance FROM accounts WHERE id = 1;   -- 여전히 10000
```

핵심은 락이 아니라 **스냅샷 생성 시점**입니다.

Phase 4. 일반 `SELECT`와 잠금 읽기는 왜 다를까?

MVCC를 이해할 때 가장 많이 헷갈리는 부분이 여기입니다. `SELECT`는 같은 읽기인데, 왜 어떤 것은 안 막히고 어떤 것은 대기할까요?

일반 `SELECT` — 스냅샷 읽기

```
SELECT balance
FROM accounts
WHERE id = 1;
```

일반 `SELECT`는 **현재 시점의 최신 값**을 무조건 읽는 것이 아니라, **내 Read View에서 허용되는 버전**을 읽습니다. 그래서 다른 트랜잭션이 행에 배타 락을 잡고 있어도, InnoDB에서는 Undo 로그를 따라가 **이전 버전**을 읽을 수 있습니다.

잠금 읽기 — 현재 버전에 대한 충돌 확인

```
SELECT balance
FROM accounts
WHERE id = 1
FOR UPDATE;
```

`FOR UPDATE`, `FOR SHARE` 같은 잠금 읽기는 단순히 "보여주는 값"만 결정하면 끝나지 않습니다. **이 행을 지금 내가 보호해야 하는가, 다른**

트랜잭션과 충돌하는가를 확인해야 하므로 현재 버전을 기준으로 락을 잡습니다.

그래서 잠금 읽기와 쓰기는 MVCC만으로 처리되지 않고, **락과 함께 동작**합니다.

읽기 종류	무엇을 읽는가	락 사용 여부
일반 `SELECT`	스냅샷에 맞는 버전	보통 락 없음
`SELECT ... FOR UPDATE`	현재 버전 + 충돌 여부 확인	행 락 사용
`SELECT ... FOR SHARE`	현재 버전 + 충돌 여부 확인	공유 락 사용

> **참고:** 그래서 "MVCC가 있으니 락이 필요 없다"는 말은 틀립니다. 정확히는 **일반 읽기를 락 없이 처리할 수 있다**가 맞습니다.

Phase 5. 실무에서 꼭 기억할 점

1. MVCC는 읽기를 공짜로 만드는 기술이 아닙니다

읽기 락을 줄여 주는 것은 맞지만, 버전을 유지하는 비용이 사라지는 것은 아닙니다. 오래 열린 트랜잭션이 많으면 오래된 버전을 정리하지 못해 **Undo 로그**가 쌓일 수 있습니다.

2. 오래 열린 트랜잭션은 생각보다 비쌉니다

사용자가 트랜잭션을 열어 둔 채 오랫동안 응답을 보내지 않거나, 배치 작업이 한 트랜잭션으로 너무 오래 실행되면 오래된 버전을 계속 붙잡게 됩니다.

- InnoDB: purge가 늦어질 수 있습니다

즉, **트랜잭션은 짧을수록 좋다**는 원칙은 락 때문만이 아니라 MVCC 때문이기도 합니다.

3. 일반 `SELECT`와 `SELECT ... FOR UPDATE`는 완전히 다릅니다

둘 다 `SELECT`라서 비슷해 보이지만, 전자는 스냅샷 읽기이고 후자는 잠금 읽기입니다. 실무에서 동시성 문제를 분석할 때 이 둘을 섞어서 보면 원인을 잘못 짚기 쉽습니다.

4. 격리 수준 문제와 락 문제를 분리해서 봐야 합니다

같은 "동시성 문제"라도

- **값이 왜 다르게 보이는가**는 MVCC/격리 수준 문제일 수 있고
- **왜 서로 대기하는가**는 락 문제일 수 있습니다

둘은 함께 동작하지만, 원인은 같지 않습니다.

정리

1. **MVCC는 여러 버전을 관리해서 읽기 일관성을 제공하는 방식입니다** — 읽기와 쓰기가 서로를 덜 막게 해줍니다
2. **InnoDB는 Undo 로그와 Read View로 MVCC를 구현합니다** — 최신 레코드만 보는 것이 아니라, 내 스냅샷에 맞는 버전을 선택합니다
3. **Read Committed와 Repeatable Read의 차이는 스냅샷 생성 시점에 있습니다** — 락보다 Read View 재생성 여부가 핵심입니다
4. **일반 `SELECT` 와 잠금 읽기는 다릅니다** — 일반 `SELECT` 는 스냅샷 읽기이고, `FOR UPDATE` 는 락을 동반하는 현재 버전 읽기입니다
5. **실무에서는 오래 열린 트랜잭션을 특히 조심해야 합니다** — 락뿐 아니라 오래된 버전 정리까지 지연시킬 수 있기 때문입니다

Chapter 5

데이터베이스 락 완전 정복 — 공유 락부터 데드락까지

#동시성 제어 #락

데이터베이스 락의 종류와 동작 원리, 낙관적·비관적 락 전략, 데드락 원인과 대처법을 예제와 함께 정리합니다.

락, 왜 알아야 하나요?

트랜잭션 격리 수준 글에서 격리 수준은 **"무엇을 보여줄지"** 를 결정하는 정책이고, 락은 **"접근 자체를 차단"** 하는 메커니즘이라고 정리했습니다. MVCC 덕분에 일반적인 읽기는 락 없이도 동작하지만, 다음과 같은 상황에서는 락이 반드시 필요합니다.

- 재고가 1개 남았는데 두 사용자가 동시에 주문합니다
- 같은 좌석을 두 명이 동시에 예약합니다
- 한 사용자가 잔액을 읽는 사이에 다른 사용자가 잔액을 변경합니다

이 글에서는 락의 종류와 범위를 먼저 살펴보고, 비관적 락과 낙관적 락의 차이, 그리고 데드락의 원인과 대처법까지 단계적으로 정리합니다.

Phase 1. 락의 기본 — 공유 락과 배타 락

모든 락의 기초가 되는 두 가지 유형입니다.

공유 락 (Shared Lock, S Lock)

읽기를 위한 락입니다. 여러 트랜잭션이 같은 데이터에 대해 동시에 공유 락을 획득할 수 있습니다.

```
SELECT * FROM accounts WHERE id = 1 FOR SHARE;
```

공유 락이 걸린 데이터는 **다른 트랜잭션도 읽을 수 있지만, 쓸 수는 없습니다.** 도서관에서 같은 책을 여러 사람이 열람할 수 있지만, 누군가 열람 중이면 책에 밑줄을 그을 수 없는 것과 비슷합니다.

> **참고:** MySQL도 버전에 따라 잠금 문법이 조금 다를 수 있습니다. 예를 들어 `FOR SHARE` 대신 `LOCK IN SHARE MODE` 문법을 사용하는 경우가 있습니다.

배타 락 (Exclusive Lock, X Lock)

쓰기를 위한 락입니다. 하나의 트랜잭션만 획득할 수 있으며, 배타 락이 걸린 데이터에는 다른 트랜잭션이 공유 락도 배타 락도 걸 수 없습니다.

```sql
-- 명시적으로 배타 락 획득
SELECT * FROM accounts WHERE id = 1 FOR UPDATE;

-- UPDATE, DELETE 문은 자동으로 배타 락 획득
UPDATE accounts SET balance = balance - 1000 WHERE id = 1;
```

호환성 정리

	공유 락 (S) 요청	배타 락 (X) 요청
공유 락 보유	허용	**대기**
배타 락 보유	**대기**	**대기**

- S + S: 여러 트랜잭션이 동시에 읽기 가능
- S + X: 읽기 중에는 쓰기 대기
- X + X: 쓰기 중에는 다른 쓰기도 대기

> **참고:** 일반 `SELECT`(스냅샷 읽기)는 MVCC를 통해 동작하므로, **InnoDB에서는 잠금 읽기와 별개로 실행됩니다.** 예를 들어 배타 락이 걸린 행이라도 일반 `SELECT`는 이전 버전을 읽어 블로킹되지 않습니다. 위 호환성 표에서 말하는 "공유 락 요청"은 `SELECT ... FOR SHARE` 같은 **잠금 읽기**를 의미합니다.

Phase 2. 락의 범위 — 공통 개념과 InnoDB 기준

락은 **무엇을 잠그느냐(잠금 대상)** 에 따라 성격이 크게 달라집니다. 이 중 **행 수준 락, 테이블 수준 락, 의도 락**은 큰 개념에서 공통적으로 이해할 수 있고, **갭 락과 넥스트 키 락**은 MySQL InnoDB 기준으로 보는 것이 안전합니다.

행 수준 락 (Row-Level Lock)

InnoDB의 기본 잠금 단위입니다. **특정 인덱스 레코드**에 락을 겁니다.

```
-- id=1 행에만 배타 락
SELECT * FROM accounts WHERE id = 1 FOR UPDATE;
```

잠금 범위가 좁아서 동시성이 높지만, 잠금 대상이 많아지면 오버헤드가 커질 수 있습니다.

갭 락 (Gap Lock, InnoDB)

인덱스 레코드 **사이의 간격**을 잠급니다. 해당 간격에 새로운 행이 삽입되는 것을 방지합니다.

```
-- age 컬럼에 인덱스가 있고, 현재 age 값이 10, 20, 30인 행이 존재한다
고 가정
SELECT * FROM users WHERE age BETWEEN 15 AND 25 FOR
UPDATE;
```

이 쿼리는 age가 10인 레코드와 20인 레코드 사이, 그리고 20인 레코드와 30인 레코드 사이의 **간격**을 잠급니다. 다른 트랜잭션이 `age = 12`나 `age = 22` 같은 행을 삽입하려고 하면 대기합니다.

```
인덱스:     10 ——— 20 ——— 30
갭 락:         (10,20)  (20,30)
                ↑ INSERT age=12 대기
                          ↑ INSERT age=22 대기
```

갭 락은 **Phantom Read를 방지하기 위한 메커니즘**입니다. MySQL InnoDB의 Repeatable Read에서 범위 검색과 스캔에 주로 사용됩니다.

> **참고:** 갭 락은 `INSERT`만 차단합니다. 갭 범위 내의 기존 행에 대한 `SELECT`나 `UPDATE`는 갭 락이 아닌 레코드 락의 영향을 받습니다. Read Committed에서는 검색/스캔에 대한 갭 락이 대부분 비활성화되고, 외래 키 검사나 중복 키 검사에는 남을 수 있습니다.

넥스트 키 락 (Next-Key Lock, InnoDB)

레코드 락 + 갭 락의 조합입니다. 특정 인덱스 레코드와 그 앞의 간격을 함께 잠급니다. InnoDB의 Repeatable Read에서 잠금 읽기와 쓰기의 기본 잠금 방식입니다.

```
인덱스:     10 ——— 20 ——— 30
넥스트 키 락: (10,20]  (20,30]
              레코드 20 + 앞 간격  레코드 30 + 앞 간격
```

넥스트 키 락 덕분에 InnoDB는 Repeatable Read에서도 **대부분의 Phantom Read를 방지**합니다.

테이블 수준 락 (Table-Level Lock)

테이블 전체를 잠급니다. DDL(`ALTER TABLE` 등) 실행 시 **메타데이터 락(MDL)** 이 자동으로 걸리며, 명시적으로 테이블 락을 획득할 수도 있습니다.

```
-- 명시적 테이블 락 (실무에서 거의 사용하지 않음)
LOCK TABLES accounts WRITE;
-- ... 작업 ...
UNLOCK TABLES;
```

InnoDB는 행 수준 락을 사용하므로 테이블 락을 직접 사용하는 경우는 드뭅니다. 다만 `ALTER TABLE` 같은 DDL은 내부적으로 메타데이터 락을 획득하므로, **운영 중에 실행하면 해당 테이블의 읽기/쓰기 쿼리가 대기할 수 있습니다.**

의도 락 (Intention Lock)

행 수준 락과 테이블 수준 락이 공존할 때 **충돌을 빠르게 감지**하기 위한 테이블 레벨 락입니다.

트랜잭션이 행에 공유 락을 걸기 전에 테이블에 **의도 공유 락(IS)** 을, 배타 락을 걸기 전에 **의도 배타 락(IX)** 을 먼저 획득합니다.

```
트랜잭션 A: SELECT * FROM accounts WHERE id = 1 FOR UPDATE;
  1) accounts 테이블에 IX 락 획득
  2) id=1 행에 X 락 획득

트랜잭션 B: LOCK TABLES accounts WRITE;
  → accounts 테이블에 이미 IX 락이 있으므로, 테이블 X 락 획득 불가 → 대기
```

의도 락이 없다면, 테이블 락을 걸려는 트랜잭션은 **모든 행을 순회하며 기존 행 락이 있는지 확인**해야 합니다. 의도 락 덕분에 테이블 레벨에서 바로 충돌 여부를 판단할 수 있습니다.

Phase 3. 비관적 락 vs 낙관적 락

지금까지 살펴본 공유 락, 배타 락 등은 모두 **데이터베이스가 제공하는 락 메커니즘**입니다. 실무에서는 이 메커니즘을 어떻게 활용할지에 대한 **전략**이 필요합니다. 크게 비관적 락과 낙관적 락, 두 가지 접근법이 있습니다.

비관적 락 (Pessimistic Lock)

"충돌이 발생할 것이다" 라고 가정하고, 데이터를 읽는 시점에 미리 락을 거는 전략입니다.

```
START TRANSACTION;

-- 1. 읽으면서 배타 락 획득
SELECT stock FROM products WHERE id = 1 FOR UPDATE;
-- → stock = 1

-- 2. 재고 차감
UPDATE products SET stock = stock - 1 WHERE id = 1;

COMMIT;
```

다른 트랜잭션이 같은 행을 `FOR UPDATE` 나 `UPDATE` 로 접근하려 하면 **현재 트랜잭션이 커밋될 때까지 대기**합니다. 데이터 정합성이 확실히 보장되지만, 대기 시간이 길어질 수 있습니다.

적합한 경우: - 충돌 빈도가 높은 경우 (인기 상품 재고 차감 등) - 데이터 정합성이 절대적으로 중요한 경우 (결제, 송금 등) - 충돌 시 재시도 비용이 큰 경우

낙관적 락 (Optimistic Lock)

"충돌이 거의 없을 것이다" 라고 가정하고, 락 없이 진행한 뒤 **업데이트 시점(예: flush 시점)에 충돌을 감지**하는 전략입니다. 데이터베이스 락을 사용하지 않고, 애플리케이션 레벨에서 **버전 번호** 또는 **타임스탬프** 컬럼으로 구현합니다.

```
-- 1. 버전과 함께 읽기 (일반 SELECT, 락 없음)
SELECT stock, version FROM products WHERE id = 1;
-- → stock = 1, version = 3

-- 2. 업데이트 시 버전 확인
UPDATE products
SET stock = stock - 1, version = version + 1
WHERE id = 1 AND version = 3;
-- → 영향받은 행: 1이면 성공, 0이면 다른 트랜잭션이 먼저 수정한 것
```

버전이 일치하지 않으면 `UPDATE`의 영향받은 행 수가 0이 됩니다. 애플리케이션은 이를 감지하여 **재시도하거나 사용자에게 알림**을 보냅니다.

JPA에서의 낙관적 락:

```
@Entity
class Product(
    @Id val id: Long,
    var stock: Int,

    @Version
    var version: Long = 0  // JPA가 자동으로 버전 관리
)
```

JPA의 `@Version` 애노테이션을 사용하면 업데이트 시 자동으로 버전을 비교하고, 충돌이 감지되면 `OptimisticLockException`을 던집니다.

적합한 경우: - 충돌 빈도가 낮은 경우 (사용자 프로필 수정 등) - 읽기가 많고 쓰기가 적은 경우 - 대기 시간보다 처리량이 중요한 경우

비교 정리

기준	비관적 락	낙관적 락
충돌 가정	충돌이 잦다고 가정	충돌이 드물다고 가정
락 시점	읽기 시점에 DB 락 획득	락 없이 진행, 업데이트 시 충돌 감지
구현 위치	데이터베이스 (FOR UPDATE)	애플리케이션 (version 컬럼)
대기	락 획득까지 블로킹	대기 없음 (충돌 시 재시도)
데드락 위험	있음	없음
적합한 상황	높은 충돌률, 정합성 최우선	낮은 충돌률, 처리량 최우선

> **참고:** 두 전략은 양자택일이 아닙니다. 하나의 서비스에서도 기능별로 다른 전략을 사용할 수 있습니다. 예를 들어 **결제 처리는 비관적 락**, **사용자 프로필 수정은 낙관적 락**을 적용하는 식입니다.
>
> 어떤 상황에서 `@Version`과 `FOR UPDATE` 중 무엇을 고를지까지 보고 싶다면, 낙관적 락 vs 비관적 락 — `@Version`과 `FOR UPDATE`를 고르는 실무 기준 글로 이어서 보는 편이 좋습니다.

Phase 4. 데드락 — 원인과 대처

데드락이란

두 개 이상의 트랜잭션이 **서로가 보유한 락을 기다리며 영원히 진행하지 못하는 상태**입니다.

```
트랜잭션 A                              트랜잭션 B
──────────────────────────────────────────────────────
UPDATE accounts
SET balance = balance - 1000
WHERE id = 1;  (행 1에 X 락)
                                        UPDATE accounts
                                        SET balance = balance
- 500
                                        WHERE id = 2;  (행 2에 X
락)
UPDATE accounts
SET balance = balance + 1000
WHERE id = 2;  → 대기 (B가 행 2 보유)
                                        UPDATE accounts
                                        SET balance = balance
+ 500
                                        WHERE id = 1;  → 대기 (A
가 행 1 보유)
→ 데드락! (A→B 대기, B→A 대기 = 순환 대기)
```

데드락 발생 조건

데드락은 다음 네 가지 조건이 **모두** 충족될 때 발생합니다.

1. **상호 배제** — 락은 한 번에 하나의 트랜잭션만 보유할 수 있습니다
2. **점유 대기** — 락을 보유한 채로 다른 락을 기다립니다
3. **비선점** — 다른 트랜잭션이 보유한 락을 강제로 빼앗을 수 없습니다
4. **순환 대기** — 트랜잭션 간 대기 관계가 원형을 이룹니다

이 중 하나라도 깨뜨리면 데드락은 발생하지 않습니다. 실무에서 가장 깨뜨리기 쉬운 조건은 **순환 대기**입니다.

DB 엔진의 데드락 처리

MySQL InnoDB: - **대기 그래프(Wait-for Graph)** 에서 순환 대기가 감지되면, Undo 로그 양이 가장 적은(롤백 비용이 가장 낮은) 트랜잭션을 선택하여 롤백합니다 - 롤백된 트랜잭션은 `ERROR 1213 (40001): Deadlock found when trying to get lock` 에러를 받습니다 - `SHOW ENGINE INNODB STATUS` 로 마지막 데드락 정보를 확인할 수 있습니다

데드락 확인 방법

```
-- MySQL: 마지막 데드락 정보 확인
SHOW ENGINE INNODB STATUS\G
-- LATEST DETECTED DEADLOCK 섹션을 확인
```

실무에서 데드락 줄이기

1. 락 획득 순서를 통일합니다

데드락의 가장 흔한 원인은 트랜잭션마다 **다른 순서로 락을 획득**하는 것입니다.

```
-- 나쁜 예: A는 1→2, B는 2→1 순서로 접근
-- 트랜잭션 A                        트랜잭션 B
-- UPDATE ... WHERE id = 1;      UPDATE ... WHERE id = 2;
-- UPDATE ... WHERE id = 2;      UPDATE ... WHERE id = 1;

-- 좋은 예: 항상 id 오름차순으로 접근
-- 트랜잭션 A                        트랜잭션 B
-- UPDATE ... WHERE id = 1;      UPDATE ... WHERE id = 1;
→ 대기
-- UPDATE ... WHERE id = 2;      UPDATE ... WHERE id = 2;
```

순서를 통일하면 **순환 대기가 원천적으로 발생하지 않습니다.**

2. 트랜잭션을 짧게 유지합니다

```
// 나쁜 예: 트랜잭션 안에서 외부 API 호출
@Transactional
fun processOrder(orderId: Long) {
    val order = orderRepository.findByIdForUpdate(orderId)
// 락 획득
    val result = externalPaymentApi.charge(order.amount)
// 수백 ms 소요
    order.complete(result)
// 락 해제는 커밋 시점
}

// 좋은 예: 외부 호출을 트랜잭션 밖으로 분리
fun processOrder(orderId: Long) {
    val order = orderRepository.findById(orderId)
    val result = externalPaymentApi.charge(order.amount)
// 락 없이 실행

    completeOrder(orderId, result)
// 짧은 트랜잭션
}

@Transactional
fun completeOrder(orderId: Long, paymentResult:
PaymentResult) {
    val order = orderRepository.findByIdForUpdate(orderId)
// 락 획득
    order.complete(paymentResult)
// 바로 커밋
}
```

락 보유 시간이 짧을수록 다른 트랜잭션과 충돌할 확률이 줄어듭니다. **트랜잭션 내에서 외부 API 호출, 파일 I/O 등 시간이 오래 걸리는 작업은 피해야 합니다.**

3. 적절한 인덱스를 사용합니다

```
-- 인덱스 없는 경우: 풀 테이블 스캔으로 더 넓은 범위를 검사
UPDATE orders SET status = 'SHIPPED' WHERE customer_id =
42;
-- customer_id에 인덱스가 없으면 많은 레코드를 검사하며 잠금 범위가 넓어
질 수 있다

-- 인덱스 있는 경우: 대상 레코드를 더 정확히 찾음
-- CREATE INDEX idx_customer_id ON orders(customer_id);
UPDATE orders SET status = 'SHIPPED' WHERE customer_id =
42;
-- customer_id = 42인 레코드 위주로 잠금이 걸린다
```

인덱스가 없으면 InnoDB는 조건에 맞는 행을 찾기 위해 **더 많은 레코드와 범위를 스캔하면서 잠금 범위가 넓어질 수 있습니다.** 이렇게 잠금 범위가 넓어지면 충돌 확률이 높아지고 데드락 위험도 커집니다.

4. 재시도 로직을 구현합니다

데드락을 완전히 제거하는 것은 어렵습니다. DB가 데드락을 감지하고 한쪽 트랜잭션을 롤백하면, 애플리케이션에서 이를 잡아 **재시도**하는 것이 현실적인 대응입니다.

```
fun executeWithRetry(maxRetries: Int = 3, action: () →
Unit) {
    var attempts = 0
    while (attempts < maxRetries) {
        try {
            action()
            return
        } catch (e: Exception) {
            if (isDeadlockException(e) && attempts <
maxRetries - 1) {
                attempts++
                Thread.sleep(50L * attempts)  // 점진적 대기
            } else {
                throw e
            }
        }
    }
}
```

한눈에 보는 InnoDB 특화 락 개념

락 종류	잠금 대상	목적
레코드 락	특정 인덱스 레코드	행 단위 읽기/쓰기 제어
갭 락	인덱스 레코드 사이의 간격	INSERT 차단 (Phantom 방지)
넥스트 키 락	레코드 + 앞 간격	InnoDB RR의 기본 잠금 방식
의도 락 (IS/IX)	테이블	행 락과 테이블 락 간 빠른 충돌 감지
테이블 락	테이블 전체	DDL, 명시적 LOCK TABLES

정리

1. **공유 락(S)과 배타 락(X)** — 모든 락의 기초입니다. S끼리는 호환되지만 X는 어떤 락과도 호환되지 않습니다
2. **일반 `SELECT`와 잠금 읽기를 구분해야 합니다** — InnoDB의 MVCC 스냅샷 읽기는 블로킹 없이 동작하지만, `FOR SHARE`, `FOR UPDATE`는 별도의 잠금 읽기입니다
3. **갭 락과 넥스트 키 락은 InnoDB 기준으로 이해해야 합니다** — Repeatable Read에서 Phantom Read를 줄이는 핵심 메커니즘이며, 인덱스 구조와 직접 연관되므로 적절한 인덱스 설계가 락 범위에 영향을 줍니다

4. **비관적 락 vs 낙관적 락** — 충돌 빈도와 정합성 요구 수준에 따라 선택합니다. 비관적 락은 DB 락(`FOR UPDATE`)을, 낙관적 락은 애플리케이션 레벨의 버전 관리를 사용합니다
5. **데드락은 완전히 제거하기 어렵습니다** — 락 획득 순서 통일, 트랜잭션 최소화, 적절한 인덱스가 예방의 핵심이며, 재시도 로직으로 대비합니다
6. **실무 판단** — 많은 경우 DB의 행 수준 락과 MVCC만으로도 충분하며, 명시적 락 전략은 동시성 문제가 실제로 발생하거나 발생 가능성이 높은 지점에 선별적으로 적용하는 것이 효과적입니다

Chapter 6

2단계 로킹 규약 완전 정복 — 2PL, Strict 2PL, 직렬 가능성까지

#동시성 제어 #트랜잭션 #락

2PL의 Growing phase와 Shrinking phase, Strict 2PL과 Conservative 2PL의 차이, 그리고 MySQL InnoDB를 2PL로만 보면 안 되는 이유를 정리합니다.

2PL, 왜 따로 알아야 하나요?

데이터베이스 락 글에서 **공유 락(S)**, **배타 락(X)**, **갭 락**, **넥스트 키 락**을 정리했고, SELECT ... FOR UPDATE 글에서는 비관적 락이 언제 필요한지도 살펴봤습니다. 그런데 락 종류를 안다고 해서 동시성 제어를 다 이해한 것은 아닙니다.

실제로는 이런 질문이 남습니다.

- 락을 **어떤 순서로 획득하고 해제해야** 안전할까요?
- 왜 어떤 스케줄은 직렬 실행한 것처럼 안전하고, 어떤 스케줄은 꼬일까요?
- `Strict 2PL`이 왜 복구와 데드락 이야기에서 자주 같이 나올까요?

이 질문에 답하는 것이 **2단계 로킹 규약(2PL, Two-Phase Locking)**입니다. 이 글에서는 교과서적인 정의만 나열하지 않고, **직렬 가능성**, **데드락**,

MySQL InnoDB의 실제 동작까지 연결해서 보겠습니다.

먼저 가장 짧은 답부터 보면

- **2PL**은 트랜잭션이 락을 다루는 과정을 **확장 단계**와 **축소 단계**로 나누는 규약입니다
- 핵심 규칙은 단순합니다. **한 번 락을 해제하기 시작하면, 그 뒤로는 새로운 락을 얻을 수 없습니다**
- 이 규약의 목적은 동시 실행 결과를 **충돌 직렬가능(conflict-serializable)** 하게 만드는 것입니다
- **Strict 2PL**은 배타 락을 커밋/롤백까지 유지해서 **cascading abort**를 막고 복구를 단순하게 만듭니다
- 다만 **2PL이 곧 데드락 방지**를 뜻하는 것은 아닙니다. 데드락은 여전히 생길 수 있습니다
- 그리고 **MySQL InnoDB는 순수한 2PL 엔진이 아닙니다.** 일반 `SELECT` 는 MVCC 스냅샷 읽기로 처리하고, 잠금 읽기와 쓰기에서 락을 적극적으로 사용합니다

Phase 1. 락 종류, 전략, 규약은 서로 다릅니다

이 셋을 섞어서 이해하면 2PL이 애매해집니다.

1. 락 종류

무엇을 어떤 방식으로 잠글지를 말합니다.

- 공유 락(S)
- 배타 락(X)
- 갭 락
- 넥스트 키 락

즉, **락의 모양**에 대한 이야기입니다.

2. 락 전략

충돌을 언제 처리할지에 대한 이야기입니다.

- 비관적 락 — 충돌이 날 것이라 보고 먼저 잠급니다
- 낙관적 락 — 끝에서 충돌을 검증합니다

즉, **운영 방식**에 대한 이야기입니다.

3. 락 규약

트랜잭션이 **언제 락을 얻고, 언제 풀 수 있는지**에 대한 규칙입니다. 2PL은 여기에 속합니다.

즉, 2PL은 "S 락을 쓸까 X 락을 쓸까"의 문제가 아니라, **락 획득과 해제의 순서를 어떻게 제한할 것인가**의 문제입니다.

Phase 2. 2PL의 핵심 규칙 — Growing phase와 Shrinking phase

2PL은 트랜잭션의 락 동작을 두 단계로 나눕니다.

확장 단계 (Growing Phase)

- 새로운 락을 획득할 수 있습니다
- 이미 가진 락의 업그레이드도 가능합니다
- 하지만 락을 해제할 수는 없습니다

축소 단계 (Shrinking Phase)

- 락을 해제할 수 있습니다
- 하지만 새로운 락을 획득할 수는 없습니다

가장 중요한 규칙은 이것입니다.

> **첫 번째 `UNLOCK` 이후에는 새로운 `LOCK`이 나오면 안 됩니다**

간단히 쓰면 이런 모양입니다.

```
확장 단계:   LOCK(A) → LOCK(B) → LOCK(C)
축소 단계:   UNLOCK(C) → UNLOCK(B) → UNLOCK(A)

허용되지 않는 형태:
LOCK(A) → UNLOCK(A) → LOCK(B)
```

마지막 예시가 바로 2PL 위반입니다. `A`를 풀기 시작한 뒤에 `B`를 새로 잡으려 했기 때문입니다.

예시로 보면 더 직관적입니다

계좌 이체를 처리하는 트랜잭션이 `from_account`, `to_account` 두 행을 모두 수정해야 한다고 가정해 보겠습니다.

```
START TRANSACTION;

SELECT * FROM accounts WHERE id = 1 FOR UPDATE;
SELECT * FROM accounts WHERE id = 2 FOR UPDATE;

UPDATE accounts SET balance = balance - 1000 WHERE id = 1;
UPDATE accounts SET balance = balance + 1000 WHERE id = 2;

COMMIT;
```

이 흐름은 보통 다음처럼 해석할 수 있습니다.

1. 먼저 필요한 락을 모두 획득합니다
2. 작업을 수행합니다
3. 트랜잭션 종료 시점에 락을 해제합니다

즉, 실무에서 많이 보는 비관적 락 패턴은 대개 **Strict 2PL에 가까운 형태**입니다.

Phase 3. 왜 2PL이면 직렬 가능성이 생길까?

교과서에서는 보통 **락 포인트(lock point)** 라는 개념으로 설명합니다.

락 포인트는 각 트랜잭션이 **마지막 락을 획득한 순간**입니다. 2PL을 지키는 트랜잭션은 이 시점을 지나면 더 이상 새 락을 잡지 못하고, 이제는 해제만 할 수 있습니다.

직관은 이렇습니다.

- 트랜잭션 A가 어떤 데이터에 필요한 락을 모두 잡았습니다
- 그 시점 이후에는 A가 갑자기 새로운 충돌 지점을 만들어 내지 못합니다
- 그래서 여러 트랜잭션의 동시 실행을 각 트랜잭션의 **락 포인트 순서**로 직렬 실행한 것처럼 재배열할 수 있습니다

이 때문에 **모든 트랜잭션이 2PL을 따르면 그 스케줄은 충돌 직렬가능**합니다.

그림으로 보면

```
트랜잭션 T1: LOCK-X(A) → LOCK-X(B) → [lock point] →
UNLOCK(B) → UNLOCK(A)
트랜잭션 T2:                        LOCK-S(C) → [lock point] →
UNLOCK(C)
```

위 스케줄에서 `T1`의 락 포인트가 `T2`보다 먼저라면, 전체 실행 결과를 **`T1` → `T2` 순서로 직렬 실행한 것과 같은 결과**로 해석할 수 있습니다.

여기서 중요한 점은, 2PL이 **실제로 한 번에 하나씩 실행한다**는 뜻은 아니라는 것입니다. 동시 실행은 유지하되, **결과가 직렬 실행과 충돌 관점에서 동일하도록** 제한하는 것입니다.

Phase 4. 2PL의 변형 — Basic, Strict, Rigorous, Conservative

실제로는 "2PL"이라고 해도 한 종류만 있는 것이 아닙니다. 자주 나오는 네 가지를 구분해 두는 편이 좋습니다.

Basic 2PL

가장 기본 규칙만 지킵니다.

- 락을 획득하는 구간과 해제하는 구간을 섞지 않습니다
- 한 번 해제를 시작하면 새 락을 잡지 않습니다

장점은 **충돌 직렬가능성**을 보장한다는 점입니다. 하지만 이것만으로는 복구가 깔끔하지 않을 수 있습니다.

예를 들어 이런 상황이 가능합니다.

```
T1: X-lock(A) → WRITE(A=100) → UNLOCK(A)
T2: S-lock(A) → READ(A=100)
T1: ROLLBACK
```

`T2`는 `T1`이 아직 커밋하지 않은 값을 읽었습니다. 만약 `T1`이 롤백되면 `T2`도 함께 문제가 될 수 있습니다. 이것이 **cascading abort** 문제입니다.

Strict 2PL

배타 락(X lock)을 트랜잭션 종료 시점까지 유지합니다.

- 새 락 획득/해제 규칙은 여전히 2PL을 따릅니다
- 여기에 더해, 쓴 데이터에 대한 X 락은 커밋/롤백 전까지 풀지 않습니다

이렇게 하면 다른 트랜잭션이 **아직 커밋되지 않은 쓰기 결과를 덮어쓰거나, 그 결과에 의존해 연쇄적으로 꼬이는 일**을 막기 쉬워집니다. 그래서:

- **Dirty Write**
- **Cascading Abort**

를 막는 데 특히 유리합니다. 그리고 **읽기도 락으로 제어하는 모델**에서는 **Dirty Read** 방지와도 자연스럽게 연결됩니다. 실무에서 "DB 락은 보통 트랜잭션 끝까지 들고 간다"는 감각은 대부분 이 계열에서 옵니다.

Rigorous 2PL

공유 락과 배타 락을 모두 트랜잭션 종료 시점까지 유지합니다.

- X 락만이 아니라 S 락도 커밋까지 유지합니다
- 가장 이해하기 쉬운 형태의 락 기반 직렬화입니다

대신 블로킹이 더 많습니다. 읽기 락도 오래 들고 있으니 동시성이 더 떨어질 수 있습니다.

Conservative 2PL

필요한 락을 시작 전에 모두 확보한 뒤에만 실행합니다. `Static 2PL`이라고도 부릅니다.

```
트랜잭션 시작 전:
LOCK-X(A), LOCK-X(B)를 모두 얻을 수 있으면 시작
둘 중 하나라도 못 얻으면 아예 시작하지 않고 대기
```

이 방식의 핵심은 **일부 락을 쥔 채 나머지 락을 기다리지 않는다**는 점입니다. 그래서 **데드락을 원천적으로 방지**할 수 있습니다.

대신 단점도 분명합니다.

- 미리 어떤 락이 필요한지 알아야 합니다
- 시작 전에 많이 기다릴 수 있습니다
- 동시성이 떨어집니다

Phase 5. 2PL이 해결하는 것과 못 하는 것

2PL은 강력하지만, 만능은 아닙니다.

1. 해결하는 것: 충돌 직렬가능성

가장 핵심적인 보장은 이것입니다.

- 여러 트랜잭션이 동시에 실행되어도

- 락 충돌 관점에서는
- 어떤 직렬 실행 순서와 같은 결과를 만들 수 있습니다

즉, **"결과가 마치 순서대로 실행한 것처럼 보이게 만든다"** 가 2PL의 핵심 가치입니다.

2. 못 하는 것: 데드락 제거

Basic 2PL이나 Strict 2PL에서도 데드락은 충분히 생길 수 있습니다.

```
트랜잭션 A                          트랜잭션 B
──────────────────────────────────────────────
LOCK-X(행 1)
                                    LOCK-X(행 2)
LOCK-X(행 2)  → 대기
                                    LOCK-X(행 1)  → 대기
```

둘 다 아직 **확장 단계**에 있습니다. 즉, 2PL을 위반한 것도 아닙니다. 그럼에도 순환 대기가 만들어져 데드락이 됩니다.

그래서 실무에서는 2PL을 안다는 것만으로는 부족하고, 데이터베이스 락 글에서 본 것처럼:

- 락 획득 순서를 통일하고
- 트랜잭션을 짧게 유지하고
- 적절한 인덱스로 잠금 범위를 좁히는 습관

이 함께 필요합니다.

3. 못 하는 것: 잘못된 잠금 대상 보완

2PL은 **언제 잠글지**에 대한 규칙이지, **무엇을 잠글지**를 자동으로 정해 주지는 않습니다.

예를 들어 팬텀 문제를 막으려면 단순히 기존 행만 잠그는 것으로는 부족하고, **검색 범위 자체**를 잠가야 할 수 있습니다. MySQL InnoDB가 트랜잭션 격리 수준과 데이터베이스 락에서 본 **갭 락**과 **넥스트 키 락**을 사용하는 이유가 여기에 있습니다.

즉, **2PL + 적절한 잠금 단위**가 함께 가야 합니다.

Phase 6. 실무에서는 어떻게 보이나?

이론을 실제 코드에 연결하면 보통 이렇게 보입니다.

1. `SELECT ... FOR UPDATE`는 Strict 2PL에 가까운 패턴입니다

```
START TRANSACTION;

SELECT * FROM products WHERE id = 1 FOR UPDATE;

-- 재고 확인
-- 비즈니스 로직 수행

UPDATE products
SET stock = stock - 1
WHERE id = 1;

COMMIT;
```

이 패턴은:

- 읽기 시점에 X 락을 잡고
- 트랜잭션 끝까지 유지한 뒤
- 커밋 시점에 락을 해제합니다

즉, 실무 감각으로 보면 **Strict 2PL 스타일의 비관적 락 사용**에 가깝습니다.

2. 여러 행을 잠글 때는 항상 순서를 맞추는 편이 좋습니다

2PL은 직렬 가능성을 높여 주지만, **락 순서를 통일해 주지는 않습니다.** 예를 들어 송금처럼 두 계좌를 동시에 잠가야 한다면 보통 `id` 오름차순으로 접근 규칙을 맞춥니다.

```
항상 작은 id를 먼저 잠금
id=1 → id=2
```

이렇게 해야 서로 반대 순서로 들어가며 데드락을 만드는 일을 줄일 수 있습니다.

3. 락으로 해결할 수 있어도, 더 짧은 SQL이 있으면 먼저 봐야 합니다

모든 경쟁 구간을 `FOR UPDATE`로 감쌀 필요는 없습니다.

```sql
UPDATE products
SET stock = stock - 1
WHERE id = 1
  AND stock > 0;
```

이런 쿼리로 규칙을 한 문장에 담을 수 있다면:

- 락 보유 시간이 짧고
- 코드가 단순하며
- 충돌 면적이 줄어듭니다

즉, 2PL은 중요한 이론이지만, **실무에서 항상 가장 좋은 구현 방식이 장시간 락 보유라는 뜻은 아닙니다.**

Phase 7. MySQL InnoDB를 2PL로만 보면 안 되는 이유

실무에서 가장 자주 헷갈리는 지점입니다.

일반 `SELECT`는 보통 S 락을 잡지 않습니다

InnoDB의 일반 `SELECT`는 대개 **MVCC 기반 스냅샷 읽기**입니다. 즉:

- 다른 트랜잭션이 쓰고 있어도
- 이전 버전을 읽고
- 블로킹 없이 진행할 수 있습니다

이 부분은 MVCC 글에서 본 것처럼 **락 기반 직렬화와는 다른 축**입니다.

잠금 읽기와 쓰기에서는 락 규약이 중요합니다

반면 다음 연산은 락을 직접 씁니다.

- `SELECT ... FOR UPDATE`
- `SELECT ... FOR SHARE`
- `UPDATE`
- `DELETE`

이 경우 InnoDB는 레코드 락, 갭 락, 넥스트 키 락 등을 잡고, 공식 문서 기준으로 **커밋이나 롤백 시점까지 유지되는 잠금**을 중심으로 동작합니다. 다만 `READ COMMITTED`에서 **수정되지 않는 행의 잠금**처럼 더 일찍 풀리는

예외도 있습니다. 이 구간은 실무적으로 **Strict 2PL에 가까운 느낌**으로 이해하는 편이 맞습니다.

그래서 InnoDB는 하이브리드로 봐야 합니다

정리하면 InnoDB는:

- **일반 읽기에는 MVCC**
- **잠금 읽기와 쓰기에는 락**

을 함께 사용합니다.

즉, "InnoDB는 2PL 엔진이다"라고만 보면 MVCC를 놓치고, 반대로 "MVCC라서 락은 중요하지 않다"라고 보면 `FOR UPDATE`, 데드락, 넥스트 키 락 문제를 놓치게 됩니다.

실무에서는 **MVCC + 락 기반 제어가 같이 동작하는 하이브리드 모델**로 이해하는 편이 가장 안전합니다.

한눈에 보는 2PL 변형

규약	핵심 규칙	장점	주의점
Basic 2PL	해제 시작 후 새 락 획득 금지	충돌 직렬가능성 보장	데드락, cascading abort 가능
Strict 2PL	X 락을 커밋/롤백까지 유지	복구 단순, Dirty Write/Cascading Abort 방지	대기 시간 증가
Rigorous 2PL	S/X 락 모두 종료 시점까지 유지	가장 이해하기 쉬운 형태	블로킹 증가
Conservative 2PL	시작 전에 필요한 락을 모두 확보	데드락 방지	필요한 락 집합을 미리 알아야 함

정리

1. **2PL은 락의 종류가 아니라 락의 사용 규칙입니다** — 한 번 풀기 시작하면 새로 잡지 않는다는 규칙이 핵심입니다
2. **핵심 목적은 충돌 직렬가능성입니다** — 동시 실행 결과를 직렬 실행과 같은 결과로 보이게 만듭니다
3. **실무에서는 Strict 2PL 감각이 더 중요합니다** — X 락을 트랜잭션 끝까지 유지해야 복구와 정합성 측면에서 다루기 쉽습니다

4. **2PL만으로 데드락이 없어지지는 않습니다** — 락 순서 통일, 짧은 트랜잭션, 적절한 인덱스가 함께 필요합니다
5. **InnoDB는 순수 2PL이 아니라 MVCC와 락의 조합입니다** — 일반 `SELECT` 와 잠금 읽기/쓰기를 분리해서 이해해야 합니다

핵심을 한 문장으로 줄이면 이렇습니다.

> **"락의 종류를 아는 것" 다음 단계는, 그 락을 어떤 순서와 규칙으로 운용해야 직렬 가능성과 복구 안정성을 얻는지 이해하는 것이고, 그 대표 규약이 바로 2PL입니다**

PART 3

3부. 락과 멱등성 실무

Chapter 7

SELECT ... FOR UPDATE는 언제 써야 할까 — 비관적 락이 필요한 순간

#동시성 제어 #트랜잭션 #락

SELECT ... FOR UPDATE가 정확히 무엇을 잠그는지, 어떤 상황에서 필요하고 어떤 상황에서는 과한지, 인덱스와 트랜잭션 범위까지 실무 기준으로 정리합니다.

`SELECT ... FOR UPDATE`, 왜 따로 알아야 하나요?

트랜잭션 격리 수준과 데이터베이스 락 글에서 락의 개념은 이미 다뤘습니다. 하지만 실무에서는 개념보다 더 자주 이런 질문을 하게 됩니다.

- 재고 차감에는 `FOR UPDATE`가 꼭 필요할까요?
- 그냥 `UPDATE stock = stock - 1`만 해도 되는 것 아닐까요?
- 읽고 계산한 뒤 저장하는 로직에서는 언제 락을 걸어야 할까요?
- `FOR UPDATE`를 붙였는데 왜 성능이 갑자기 떨어질까요?

`SELECT ... FOR UPDATE`는 단순히 "안전하게 만드는 마법 문장"이 아닙니다. **읽는 순간 락을 잡아서 이후 쓰기까지 그 상태를 보호**하고 싶을 때 쓰는 도구입니다. 정확히는 **읽기-판단-쓰기** 사이에 다른 트랜잭션이 끼어들면 안 되는 경우에 의미가 있습니다.

이 글에서는 MySQL InnoDB 기준으로 `SELECT ... FOR UPDATE`가 언제 필요한지, 언제는 오히려 과한지, 그리고 실무에서 어떤 점을 조심해야 하는지 정리합니다.

`FOR UPDATE` 자체보다 더 큰 선택, 즉 **언제 비관적 락을 쓰고 언제 `@Version` 같은 낙관적 락을 택할지**까지 비교하고 싶다면 낙관적 락 vs 비관적 락 — `@Version`과 `FOR UPDATE`를 고르는 실무 기준 글을 이어서 보면 자연스럽습니다.

Phase 1. `SELECT ... FOR UPDATE`는 정확히 무엇을 하는가?

가장 단순한 예시는 다음과 같습니다.

```
START TRANSACTION;

SELECT *
FROM products
WHERE id = 1
FOR UPDATE;

UPDATE products
SET stock = stock - 1
WHERE id = 1;

COMMIT;
```

이 쿼리는 `id = 1` 행을 읽으면서 **배타 락(X Lock)**을 획득합니다. 그 결과 다른 트랜잭션은 이 행에 대해:

- `UPDATE`, `DELETE`를 바로 수행할 수 없고
- `SELECT ... FOR UPDATE`, `SELECT ... FOR SHARE` 같은 잠금 읽기도 대기하게 됩니다

즉, 핵심은 **지금 읽은 행을 내가 트랜잭션 끝날 때까지 보호하겠다**는 선언입니다.

일반 `SELECT`와의 차이

```
-- 일반 SELECT
SELECT * FROM products WHERE id = 1;

-- 잠금 읽기
SELECT * FROM products WHERE id = 1 FOR UPDATE;
```

일반 `SELECT`는 MVCC 스냅샷 읽기이므로 다른 트랜잭션의 쓰기를 막지 않습니다. 반면 `FOR UPDATE`는 **현재 버전에 락을 걸기 때문에**, 이후 수정 경쟁을 직접 제어할 수 있습니다.

Phase 2. 언제 꼭 필요할까?

`FOR UPDATE`가 빛나는 순간은 대부분 비슷합니다. **읽은 값을 기준으로 애플리케이션이 판단을 내린 뒤, 그 결과를 다시 쓰는 경우**입니다.

1. 재고 차감

가장 대표적인 예시입니다.

```sql
START TRANSACTION;

SELECT stock
FROM products
WHERE id = 1
FOR UPDATE;

-- 애플리케이션 로직
-- stock > 0 이면 주문 가능

UPDATE products
SET stock = stock - 1
WHERE id = 1;

COMMIT;
```

여기서 락이 없으면 두 요청이 동시에 `stock = 1`을 읽고 둘 다 주문 가능하다고 판단할 수 있습니다. 그 결과 **재고가 음수로 내려가거나, 실제보다 많이 판매되는 문제**가 생깁니다.

실무에서는 이 로직이 보통 이런 형태로 들어갑니다.

```
@Transactional
fun decreaseStock(productId: Long, quantity: Int) {
    val product =
productRepository.findByIdForUpdate(productId)

    require(product.stock ⩾ quantity) {
        "재고가 부족합니다."
    }

    product.stock -= quantity
}
```

핵심은 `재고 확인`과 `재고 차감`이 하나의 트랜잭션 안에서, 같은 행에 대한 락을 잡은 상태로 이어져야 한다는 점입니다.

2. 중복 처리 방지

예를 들어 같은 쿠폰을 두 번 사용하면 안 되는 상황을 생각해 보겠습니다.

```sql
START TRANSACTION;

SELECT used
FROM coupons
WHERE coupon_id = 100
FOR UPDATE;

-- used = false 인지 확인

UPDATE coupons
SET used = true
WHERE coupon_id = 100;

COMMIT;
```

락이 없으면 두 요청이 동시에 `used = false`를 보고 둘 다 성공 처리할 수 있습니다.

3. 상태 전이(state transition) 보호

주문 상태가 `READY → PAID → SHIPPED` 순서로만 바뀌어야 한다면, 상태를 읽고 검증한 뒤 변경하는 과정도 보호가 필요할 수 있습니다.

```sql
START TRANSACTION;

SELECT status
FROM orders
WHERE id = 1
FOR UPDATE;

-- READY 상태인지 확인

UPDATE orders
SET status = 'PAID'
WHERE id = 1;

COMMIT;
```

이 패턴의 핵심은 단순 업데이트가 아니라, **현재 상태를 보고 비즈니스 판단을 한 뒤 다음 상태로 넘긴다**는 점입니다.

다만 상태 전이도 항상 `FOR UPDATE`가 필요한 것은 아닙니다. 조건이 단순하다면 한 문장 `UPDATE`로 표현할 수 있습니다.

```sql
UPDATE orders
SET status = 'PAID'
WHERE id = 1
  AND status = 'READY';
```

이 경우:

- 영향받은 행 수가 1이면 상태 변경 성공

- 0이면 이미 다른 상태로 바뀌었거나, 선행 조건을 만족하지 않은 것

즉, **현재 상태가 특정 값일 때만 변경한다**는 규칙은 원자적 `UPDATE`로 충분한 경우가 많습니다.

Phase 3. 반대로, 꼭 필요하지 않은 경우도 많습니다

실무에서 흔한 실수는 `FOR UPDATE`를 너무 넓게 쓰는 것입니다.

1. 원자적 UPDATE만으로 충분한 경우

잔액 차감 자체만 목적이고, 중간에 별도 비즈니스 판단이 없다면 다음 쿼리는 굳이 먼저 읽을 필요가 없습니다.

```
UPDATE accounts
SET balance = balance - 3000
WHERE id = 1;
```

이 쿼리는 DB가 한 문장 안에서 최신 값을 기준으로 계산합니다. 이런 경우는 별도로 `SELECT ... FOR UPDATE`를 하지 않아도 됩니다. 다만 "잔액이 부족하면 실패해야 한다" 같은 조건이 있으면 `WHERE balance ⩾ 3000`처럼 조건까지 함께 넣어야 합니다.

더 안전하게 조건까지 함께 넣을 수도 있습니다.

```
UPDATE products
SET stock = stock - 1
WHERE id = 1
  AND stock > 0;
```

이렇게 하면:

- 재고가 있을 때만 차감되고
- 영향받은 행 수가 1이면 성공
- 0이면 품절로 판단할 수 있습니다

즉, **읽고 판단하는 로직을 SQL 한 문장으로 밀어 넣을 수 있다면,** `FOR UPDATE` 가 필요 없는 경우가 많습니다.

애플리케이션에서는 보통 이렇게 해석합니다.

```
val updatedRows =
productRepository.decreaseStockIfAvailable(productId)

if (updatedRows == 0) {
    throw IllegalStateException("품절입니다.")
}
```

이 패턴의 장점은 락을 오래 쥐고 있지 않으면서도, 성공/실패를 **영향받은 행 수**로 명확하게 판단할 수 있다는 점입니다.

2. 단순 조회만 하는 경우

조회 결과를 화면에 보여 주기만 하고, 그 값을 기준으로 즉시 쓰기 결정을 하지 않는다면 `FOR UPDATE` 는 과합니다.

```
-- ❌ 단순 상세 조회에 잠금 읽기 사용
SELECT * FROM products WHERE id = 1 FOR UPDATE;
```

이런 코드는 불필요하게 락을 오래 잡아 다른 요청을 막을 수 있습니다.

3. 긴 작업과 함께 묶는 경우

```
@Transactional
fun issueCoupon(userId: Long, couponId: Long) {
    val coupon = couponRepository.findForUpdate(couponId)
    val result = externalApi.call()
    coupon.issueTo(userId)
}
```

여기서 외부 API 호출이 느리면, 락도 그만큼 오래 유지됩니다. `FOR UPDATE` 는 짧고 강하게 써야지, 오래 들고 있으면 거의 항상 문제가 됩니다.

Phase 4. 가장 중요한 판단 기준은 "읽기-판단-쓰기"인가?

`FOR UPDATE` 가 필요한지 판단하는 가장 간단한 질문은 이것입니다.

> **내가 읽은 값을 기준으로 애플리케이션이 판단을 내리고, 그 사이 다른 트랜잭션이 값을 바꾸면 안 되는가?**

`YES` 라면 `FOR UPDATE` 를 검토할 이유가 있습니다.
`NO` 라면 보통 다른 방법이 더 단순합니다.

필요한 경우

- 현재 재고를 보고 주문 가능 여부를 판단
- 현재 상태를 보고 상태 전이 가능 여부를 판단
- 현재 사용 여부를 보고 중복 처리 여부를 판단

불필요한 경우

- 단순 조회
- 읽지 않고 바로 원자적 `UPDATE`
- 결과가 조금 바뀌어도 비즈니스상 문제 없는 통계성 처리

Phase 5. 인덱스가 없으면 생각보다 넓게 잠글 수 있습니다

`FOR UPDATE` 를 쓸 때 가장 자주 놓치는 것이 인덱스입니다.

```
SELECT *
FROM orders
WHERE customer_id = 42
FOR UPDATE;
```

`customer_id`에 인덱스가 없다면 InnoDB는 조건에 맞는 행을 찾기 위해 더 넓은 범위를 스캔해야 합니다. 이 과정에서:

- 잠금 대상이 불필요하게 넓어질 수 있고
- 다른 트랜잭션 충돌이 급격히 늘고
- 데드락 가능성도 높아질 수 있습니다

즉, `FOR UPDATE`는 문장 하나만 보는 것이 아니라 **어떤 인덱스로 어떤 범위를 읽는가**까지 함께 봐야 합니다.

실무에서 확인할 것

```
EXPLAIN
SELECT *
FROM orders
WHERE customer_id = 42
FOR UPDATE;
```

`EXPLAIN` 결과가 풀 스캔에 가깝다면, 락 범위도 의도보다 커질 가능성을 의심해야 합니다.

Phase 6. 범위 조건에서는 갭 락과 넥스트 키 락도 같이 생각해야 합니다

데이터베이스 락 글에서 다뤘듯이, InnoDB는 Repeatable Read에서 범위 잠금 읽기에 **갭 락**과 **넥스트 키 락**을 함께 사용할 수 있습니다.

```
SELECT *
FROM reservations
WHERE room_id = 10
  AND reserved_at BETWEEN '2026-04-08 10:00:00' AND '2026-
04-08 11:00:00'
FOR UPDATE;
```

이런 쿼리는 단순히 현재 있는 행만 잠그는 것이 아니라, **그 범위에 새 행이 들어오는 것까지 막는 방향**으로 동작할 수 있습니다.

이게 필요한 경우도 있습니다.

- 같은 시간대 예약 중복 방지
- 특정 범위 내 선점 처리

하지만 잘못 쓰면 예상보다 훨씬 넓은 충돌을 만들 수 있습니다. 범위 잠금은 편리하지만, 그만큼 조심해서 써야 합니다.

Phase 7. `FOR UPDATE`를 쓰면 생기는 부작용

안전성을 높이는 대신 반드시 비용을 치릅니다.

1. 대기 시간 증가

한 요청이 락을 잡고 있으면 다음 요청은 기다려야 합니다.

```
요청 A: SELECT ... FOR UPDATE  → 락 획득
요청 B: SELECT ... FOR UPDATE  → 대기
```

동시 요청이 많은 구간에서는 이 대기 시간이 곧 응답 지연으로 이어집니다.

2. 데드락 위험 증가

두 트랜잭션이 서로 다른 순서로 여러 행을 잠그면 데드락이 발생할 수 있습니다.

```
트랜잭션 A: id=1 잠금 → id=2 잠금 시도
트랜잭션 B: id=2 잠금 → id=1 잠금 시도
```

이 상황에서는 한쪽이 반드시 롤백됩니다.

2-1. `lock wait timeout`과 `deadlock`은 다릅니다

둘 다 실패처럼 보이지만 의미는 다릅니다.

- `lock wait timeout` — 누군가 잡고 있는 락을 오래 기다렸지만 끝내 얻지 못한 상황
- `deadlock` — 서로가 서로를 기다리는 순환 대기가 감지되어 DB가 한쪽을 강제로 중단한 상황

실무에서는 둘 다 재시도 후보가 될 수 있지만, **`deadlock`은 락 획득 순서 문제**, **`lock wait timeout`은 긴 트랜잭션이나 느린 작업 문제**일 가능성이 더 큽니다.

3. 커넥션 점유 시간 증가

DB 커넥션 풀 글과 연결되는 부분입니다. 트랜잭션 안에서 `FOR UPDATE`를 사용하면, 락뿐 아니라 **커넥션도 그 시간 동안 점유**됩니다. 그래서:

- 느린 외부 API 호출
- 긴 서비스 로직
- 사용자 입력 대기

같은 작업과 함께 묶으면 커넥션 풀까지 함께 흔들릴 수 있습니다.

Phase 8. 실무에서 추천하는 사용 원칙

복잡하게 외우기보다 다음 원칙으로 판단하면 충분합니다.

원칙 1. 정말 필요한 행만 잠급니다

조건을 좁게 쓰고, 인덱스를 맞추고, 불필요한 범위 잠금을 피해야 합니다.

원칙 2. 트랜잭션은 최대한 짧게 유지합니다

락을 잡은 뒤 외부 API 호출, 파일 I/O, 복잡한 계산을 넣지 않는 것이 기본입니다.

원칙 3. 원자적 UPDATE로 대체 가능하면 먼저 그 방법을 봅니다

`FOR UPDATE`는 강력하지만 무겁습니다. 한 문장 `UPDATE`로 해결되면 그쪽이 더 단순하고 안전한 경우가 많습니다.

특히:

- "조건을 만족하면 차감"
- "아직 처리되지 않았으면 처리 완료로 변경"
- "현재 상태가 READY일 때만 PAID로 변경"

같은 로직은 `WHERE` 조건을 잘 설계하면 `FOR UPDATE` 없이 해결할 수 있는 경우가 많습니다.

원칙 4. 데드락 재시도 전략을 준비합니다

`FOR UPDATE`를 쓰는 구간은 충돌 가능성이 높은 구간인 경우가 많습니다. 따라서 데드락이나 `lock wait timeout`을 만났을 때 재시도 전략까지 함께 설계하는 편이 현실적입니다.

정리

1. **`SELECT ... FOR UPDATE`는 읽는 순간 행을 보호하기 위한 비관적 락입니다** — 읽기-판단-쓰기 사이를 안전하게 만들고 싶을 때 의미가 있습니다
2. **재고 차감, 중복 처리, 상태 전이처럼 현재 값을 보고 결정하는 로직에서 특히 유용합니다**

3. **원자적 `UPDATE` 로 해결 가능한 경우에는 굳이 `FOR UPDATE` 가 필요하지 않을 수 있습니다**
4. **인덱스 없이 쓰면 잠금 범위가 커질 수 있습니다** — `EXPLAIN` 으로 실제 접근 범위를 같이 봐야 합니다
5. **`FOR UPDATE` 는 안전성과 맞바꿔 대기, 데드락, 커넥션 점유 비용을 늘립니다** — 짧고 좁게 쓰는 것이 핵심입니다

큰 그림에서 `FOR UPDATE` 와 `@Version` 을 어떻게 나눠 쓸지까지 보고 싶다면, 다음 글인 낙관적 락 vs 비관적 락 — `@Version` 과 `FOR UPDATE` 를 고르는 실무 기준으로 이어서 읽으면 좋습니다.

Chapter 8

낙관적 락 vs 비관적 락 — `@Version`과 `FOR UPDATE`를 고르는 실무 기준

#동시성 제어 #트랜잭션 #락

낙관적 락과 비관적 락을 충돌률, 재시도 비용, 트랜잭션 길이 관점에서 비교하고, 조건부 `UPDATE`나 `UNIQUE` 제약이 더 나은 경우까지 정리합니다.

낙관적 락과 비관적 락, 왜 알아야 하나요?

데이터베이스 락 글에서는 락의 종류와 기본 동작을, SELECT ... FOR UPDATE 글에서는 비관적 락을 언제 써야 하는지를 따로 정리했습니다. 그런데 실제 서비스 코드에서는 여전히 이런 판단이 남습니다.

- 사용자 프로필 수정은 `@Version`이면 충분해 보이는데, 재고 차감도 같은 방식으로 처리해도 될까요?
- `FOR UPDATE`를 걸면 안전해 보이지만, 모든 쓰기 경쟁 구간에 붙여도 괜찮을까요?
- 조건부 `UPDATE`나 `UNIQUE` 제약으로 끝낼 수 있는 문제에 락 전략을 과하게 쓰고 있지는 않을까요?

핵심은 "무슨 락 문법을 쓸까?"가 아닙니다. **충돌을 언제 감지하고, 실패를 어떤 형태로 드러낼 것인가**를 먼저 정해야 합니다.

이 글은 MySQL InnoDB + Spring Boot + JPA를 기준으로, 기본 개념 설명보다 **선택 기준**에 집중합니다.

먼저 선택 기준부터 보면

실무에서는 보통 아래 순서로 판단하면 덜 헷갈립니다.

1. **SQL 한 문장으로 규칙을 표현할 수 있나?** 가능하면 락 전략보다 먼저 검토합니다
2. **같은 행에 요청이 얼마나 자주 몰리나?** 충돌이 잦으면 낙관적 락은 실패와 재시도가 급격히 늘 수 있습니다
3. **충돌 시 다시 시도해도 안전한가?** 안전하면 낙관적 락이 잘 맞고, 아니면 비관적 락이나 제약 조건이 더 낫습니다
4. **트랜잭션을 짧게 유지할 수 있나?** 길다면 비관적 락은 대기와 커넥션 점유 비용이 커집니다

가장 짧게 줄이면 이렇습니다.

- **충돌이 드물고 재시도가 자연스럽다** → 낙관적 락
- **충돌이 잦고 한 번에 한 명만 처리해야 한다** → 비관적 락
- **조건을 SQL로 바로 표현할 수 있다** → 조건부 `UPDATE`나 `UNIQUE` 제약을 먼저 검토

Phase 1. 핵심 차이는 "기다리게 할지, 실패하게 할지"입니다

두 전략의 가장 큰 차이는 구현 문법이 아니라 **충돌을 처리하는 시점**입니다.

- **비관적 락** — 충돌이 날 것이라 보고 먼저 잠급니다
- **낙관적 락** — 충돌이 드물 것이라 보고 끝에서 검증합니다

그래서 같은 쓰기 경쟁도 사용자에게는 전혀 다르게 보입니다.

- 비관적 락은 보통 **대기**, `lock wait timeout`, `deadlock`으로 나타납니다
- 낙관적 락은 보통 **충돌 감지**, 재시도, "다른 사용자가 먼저 수정했습니다" 같은 실패로 나타납니다

즉, 질문은 "어느 쪽이 더 안전한가?"가 아니라 **"이 구간의 실패를 대기로 보여 줄 것인가, 충돌 에러로 보여 줄 것인가?"** 입니다.

Phase 2. 낙관적 락이 잘 맞는 경우

낙관적 락은 **대부분의 요청이 서로 부딪히지 않는 상황**에서 강합니다.

1. 같은 레코드를 동시에 수정할 확률이 낮다

사용자 프로필 수정, 관리자 설정 변경, 게시글 편집처럼 같은 행을 동시에 건드릴 가능성이 낮은 기능은 매번 DB 락을 잡는 편이 더 과할 수 있습니다.

이럴 때는:

- 대부분의 요청은 대기 없이 바로 처리되고
- 드물게 충돌한 요청만 예외 처리하면 되므로

전체 처리량과 응답성이 더 좋아지기 쉽습니다.

2. 읽기와 쓰기 사이 간격이 길다

웹 화면에서는 사용자가 수정 화면을 열고 저장 버튼을 누르기까지 수초에서 수 분이 걸릴 수 있습니다.

```
1. 사용자 A가 수정 화면 열기
2. 내용을 5분 동안 수정
3. 저장 버튼 클릭
```

이 구간을 DB 락으로 보호하는 것은 현실적으로 어렵습니다. HTTP 요청 사이에 락을 계속 들고 있을 수 없기 때문입니다. 이런 종류의 편집 화면은 **저장 시점에 버전을 비교하는 모델**이 자연스럽습니다.

3. 충돌 시 재시도나 사용자 안내가 가능하다

낙관적 락은 충돌이 나면 애플리케이션이 후속 행동을 결정해야 합니다.

- 최신 데이터를 다시 읽고 자동 재시도
- 충돌 사실을 알려주고 다시 편집 유도
- 변경 내용을 병합

즉, 낙관적 락은 **충돌 이후의 UX까지 설계할 수 있을 때** 잘 맞습니다.

JPA에서는 보통 `@Version`으로 구현합니다

```
@Entity
class Article(
    @Id
    val id: Long,

    var title: String,
    var content: String,

    @Version
    var version: Long = 0,
)
```

`@Version`이 붙어 있으면 JPA provider는 보통 `flush` / `commit` 과정에서 실행되는 `UPDATE`에 버전 조건을 포함합니다.

```
UPDATE article
SET title = ?, content = ?, version = version + 1
WHERE id = ?
  AND version = ?;
```

영향받은 행 수가 0이 되면 충돌로 보고 `OptimisticLockException` 계열 예외를 던집니다.

> **참고:** 낙관적 락은 "오래 잠가 두는 락"이 아닙니다. 일반 조회로 읽고, 저장 시점에 내가 봤던 버전이 아직 유효한지만 확인합니다.

Phase 3. 비관적 락이 잘 맞는 경우

비관적 락은 **부딪힐 가능성이 높은 구간을 아예 줄 세우는 전략**입니다.

1. 같은 행에 요청이 자주 몰린다

인기 상품 재고 차감, 좌석 선점, 쿠폰 사용 처리처럼 같은 행이 반복해서 갱신되는 구간은 낙관적 락으로 돌리면 실패 요청이 빠르게 늘 수 있습니다.

예를 들어 재고 1개짜리 상품에 요청이 동시에 몰리면:

- 낙관적 락은 읽기까지는 여러 요청이 통과한 뒤 마지막에 대부분 실패하고
- 비관적 락은 처음부터 한 명씩 들어가도록 대기시킵니다

이런 구간에서는 "빠른 실패"보다 **예측 가능한 직렬화**가 더 중요할 수 있습니다.

2. 읽고 판단한 뒤 쓰는 로직을 통째로 보호해야 한다

다음과 같은 로직은 단순 증가/감소보다 **현재 상태에 대한 비즈니스 판단**이 핵심입니다.

- 재고가 충분한지 확인한 뒤 차감

- 쿠폰이 아직 사용되지 않았는지 확인한 뒤 사용 처리
- 주문 상태가 `READY`인지 확인한 뒤 `PAID`로 전이

이런 경우에는 SELECT ... FOR UPDATE에서 정리한 것처럼 **읽기-판단-쓰기 구간** 자체를 보호해야 할 수 있습니다.

```
START TRANSACTION;

SELECT stock
FROM products
WHERE id = 1
FOR UPDATE;

-- stock > 0 인지 확인

UPDATE products
SET stock = stock - 1
WHERE id = 1;

COMMIT;
```

3. 충돌 후 재시도 비용이 크다

비관적 락은 "충돌이 잦다"는 이유만으로 선택하는 것이 아닙니다. **실패 후 다시 시도하기 어려운 구간**에서도 자주 고려합니다.

예를 들어:

- 결제 승인 요청과 얽힌 상태 변경
- 외부 시스템 차감과 함께 가는 처리

- 중복 발송이 위험한 알림/메시지 처리

이런 구간은 충돌 후 재시도를 무심코 붙이면 부작용이 커질 수 있습니다. 물론 그렇다고 트랜잭션 안에서 외부 API를 호출하라는 뜻은 아닙니다. 비관적 락을 쓰더라도 **트랜잭션은 짧게** 유지해야 합니다.

Phase 4. 둘 중 하나를 고르기 전에 더 단순한 해결책부터 봐야 합니다

실무에서 자주 놓치는 지점입니다. 많은 문제는 낙관적 락과 비관적 락 중 하나를 고르기 전에 **SQL 한 문장**이나 **제약 조건**으로 끝낼 수 있습니다.

1. 조건부 `UPDATE`로 끝낼 수 있다

재고 차감은 자주 이렇게 표현할 수 있습니다.

```
UPDATE products
SET stock = stock - 1
WHERE id = 1
  AND stock > 0;
```

이 방식은:

- 재고가 있을 때만 차감되고
- 품절이면 아무 행도 수정하지 않으며
- 영향받은 행 수만 보면 성공/실패를 판단할 수 있습니다

즉, 규칙을 SQL 한 문장으로 안전하게 표현할 수 있다면 `FOR UPDATE`나 `@Version`보다 더 직접적일 수 있습니다.

2. 상태 전이도 조건부 `UPDATE`로 표현할 수 있다

```sql
UPDATE orders
SET status = 'PAID'
WHERE id = 1
  AND status = 'READY';
```

이 경우도 영향받은 행 수가:

- `1`이면 상태 변경 성공
- `0`이면 이미 다른 상태이거나 선행 조건 불만족

즉, "현재 값이 이럴 때만 바꾼다"는 규칙은 읽기-판단-쓰기로 풀지 않아도 되는 경우가 많습니다.

3. 중복 방지는 `UNIQUE` 제약이 더 강력할 수 있다

같은 사용자가 같은 쿠폰을 두 번 발급받으면 안 된다면, 고유 제약 조건이 락 전략보다 더 강한 해결책일 수 있습니다.

```sql
ALTER TABLE issued_coupon
ADD CONSTRAINT uk_coupon_user UNIQUE (coupon_id, user_id);
```

그 후 그냥 `INSERT`하고 중복 키 예외를 처리하면 됩니다. 이 문제의 핵심은 "잠그는 것"이 아니라 **중복을 물리적으로 허용하지 않는 것**이기 때문입니다.

Phase 5. JPA와 Spring에서 선택을 망치는 함정

실무에서는 전략 자체보다 구현 디테일 때문에 의도와 다른 동작이 더 자주 나옵니다.

1. `@Version`을 붙였는데 벌크 `UPDATE`를 사용한다

JPA의 벌크 `UPDATE`는 영속성 컨텍스트를 우회합니다. 그래서 기본적으로 `@Version` 기반 낙관적 락 검사가 자동 적용되지 않습니다.

```
@Modifying
@Query("""
    update Product p
    set p.stock = p.stock - 1
    where p.id = :id
""")
fun decreaseStock(id: Long): Int
```

이런 쿼리는 빠를 수 있지만, `version` 조건이 없으면 낙관적 락 보호를 받지 못합니다.

2. 낙관적 락 예외 시점을 잘못 기대한다

낙관적 락 예외는 보통 엔티티를 읽을 때가 아니라 `flush` 또는 `commit` 시점에 자주 발생합니다. 서비스 메서드 앞부분에서 외부 로직을 많이 수행한 뒤 마지막에 예외가 터지면 흐름이 꼬일 수 있습니다.

3. 비관적 락을 잡고 트랜잭션 안에서 오래 머문다

```
@Transactional
fun processOrder(orderId: Long) {
    val order = orderRepository.findByIdForUpdate(orderId)
    val paymentResult = externalApi.charge(order.amount)
    order.complete(paymentResult)
}
```

이 패턴은 락을 잡은 채 외부 API를 기다립니다. 응답 지연이 생기면 락 보유 시간, 커넥션 점유 시간, 대기 요청 수가 함께 늘어납니다.

4. 인덱스 없이 `FOR UPDATE`를 사용한다

인덱스가 부실하면 InnoDB는 더 넓은 범위를 스캔하고 잠글 수 있습니다. 그러면 원래 의도보다 많은 행이 잠기고, 충돌률과 `deadlock` 가능성도 함께 올라갑니다.

한눈에 보는 선택 기준

실무 판단을 표로 줄이면 아래와 가깝습니다.

질문	낙관적 락이 잘 맞는 경우	비관적 락이 잘 맞는 경우	더 먼저 볼 대안
충돌 빈도는 어떤가?	같은 행 수정이 드뭅니다	같은 행에 요청이 자주 몰립니다	-
요청 사이 간격은 어떤가?	읽기와 저장 사이 간격이 깁니다	짧은 트랜잭션 안에서 끝납니다	-
실패 후 재시도가 안전한가?	재시도나 사용자 안내가 자연스럽습니다	재시도 비용이 큽니다	`UNIQUE`, 조건부 `UPDATE`
로직이 읽기-판단-쓰기인가?	꼭 그렇지 않습니다	현재 상태를 읽고 판단해야 합니다	조건을 SQL로 밀어 넣기
부하가 몰리면 어떤 형태가 나은가?	일부 충돌 실패를 감수할 수 있습니다	대기시키더라도 직렬화가 낫습니다	큐/직렬 처리 구조 검토

정리

1. **낙관적 락과 비관적 락의 차이는 충돌을 언제 처리하느냐입니다** — 하나는 나중에 검증하고, 다른 하나는 먼저 막습니다
2. **충돌이 드물고 재시도가 자연스러우면 낙관적 락이 잘 맞습니다** — 편집 화면, 설정 변경, 낮은 충돌률의 관리 기능이 대표적입니다
3. **충돌이 잦고 읽기-판단-쓰기를 통째로 보호해야 하면 비관적 락이 잘 맞습니다** — 재고, 쿠폰, 좌석, 상태 전이 같은 구간이 대표적입니다

4. **많은 경우 더 좋은 해답은 조건부 `UPDATE`나 `UNIQUE` 제약입니다** — 락 전략은 항상 첫 선택지가 아닙니다
5. **JPA 구현 디테일이 선택을 망칠 수 있습니다** — 벌크 `UPDATE`, 늦은 예외 시점, 긴 트랜잭션, 인덱스 없는 `FOR UPDATE`는 대표적인 함정입니다

핵심을 한 문장으로 줄이면 이렇습니다.

> **어떤 락을 쓸지보다, 충돌을 대기로 처리할지 실패로 처리할지부터 정하는 편이 실무에서는 더 중요합니다**

Chapter 9

분산 락은 언제 써야 할까 — DB 락으로 충분한 경우와 Redis 락이 필요한 경우

#동시성 제어 #락

DB 행 락, MySQL `GET_LOCK()`, Redis 분산 락이 각각 어떤 범위를 보호하는지 비교하고, 조건부 `UPDATE`나 `UNIQUE` 제약이 더 나은 경우까지 정리합니다.

분산 락, 왜 알아야 하나요?

데이터베이스 락, SELECT ... FOR UPDATE, 낙관적 락 vs 비관적 락 글까지 읽고 나면 다음 질문이 자연스럽게 나옵니다.

- 재고 차감 같은 문제도 Redis 분산 락으로 풀어야 할까요?
- 여러 서버 인스턴스에서 같은 스케줄러가 동시에 돌면 DB 락으로 막을 수 있을까요?
- `FOR UPDATE`, `GET_LOCK()`, Redis 락은 무엇이 다른가요?
- 사실 락보다 `UNIQUE` 제약이나 조건부 `UPDATE`가 더 나은 경우가 있지 않을까요?

핵심은 **분산 락이 더 강한 락이냐**가 아닙니다. 먼저 구분해야 할 것은 **무엇을 보호하려는가** 입니다.

- 같은 DB 행을 수정하는 경쟁인가?

- 같은 작업이 여러 인스턴스에서 중복 실행되는 문제인가?
- 외부 API 호출, 캐시 재생성, 배치 작업처럼 DB 바깥 자원을 조율해야 하는가?

이 글은 MySQL + Redis를 기준으로, **DB 락으로 충분한 경우와 Redis 락이 필요한 경우의 경계**를 정리합니다.

먼저 선택 기준부터 보면

실무에서는 보통 아래 순서로 판단하면 크게 틀리지 않습니다.

상황	먼저 볼 선택지	이유
같은 DB 행의 상태를 읽고 판단한 뒤 수정	`FOR UPDATE` 또는 조건부 `UPDATE`	문제 범위가 이미 DB 트랜잭션 안에 있습니다
중복 삽입 방지	`UNIQUE` 제약 조건	락보다 더 직접적으로 중복을 금지할 수 있습니다
여러 앱 인스턴스가 같은 작업을 중복 실행	MySQL `GET_LOCK()` 또는 Redis 락	보호 대상이 특정 행이 아니라 작업 자체입니다
보호 대상이 DB 밖의 자원	Redis 락 같은 외부 조율 수단	DB 행 락만으로는 외부 자원을 잠글 수 없습니다
간헐적 캐시 재생성, 배치 단일 실행	`named lock` / `distributed lock` 검토	같은 "행"이 아니라 "작업 키"를 기준으로 조율해야 합니다

가장 짧게 줄이면 이렇습니다.

- **문제가 DB 행 경쟁이라면 DB 락부터 봅니다**
- **문제가 여러 프로세스의 중복 실행이라면 `named lock`이나 `distributed lock`을 봅니다**
- **락 없이 제약 조건이나 원자적 SQL로 끝나면 그쪽이 더 낫습니다**

Phase 1. DB 락과 분산 락은 애초에 푸는 문제가 다릅니다

이 둘을 같은 선상에서 비교하면 자꾸 헷갈립니다.

DB 행 락은 "데이터 변경"을 보호합니다

예를 들어 재고 차감은 이런 문제입니다.

```
START TRANSACTION;

SELECT stock
FROM products
WHERE id = 1
FOR UPDATE;

-- stock > 0 인지 확인

UPDATE products
SET stock = stock - 1
WHERE id = 1;

COMMIT;
```

여기서 보호 대상은 **`products.id = 1` 이라는 행의 현재 상태**입니다. 읽기-판단-쓰기 전체가 DB 트랜잭션 안에 있으므로, DB 락이 가장 자연스럽습니다.

분산 락은 "여러 프로세스의 실행"을 조율합니다

반면 이런 문제는 성격이 다릅니다.

- 서버 4대가 같은 스케줄러를 동시에 실행
- 같은 캐시 키 재생성이 여러 인스턴스에서 한꺼번에 시작
- 같은 외부 API 요청을 여러 워커가 동시에 발송

여기서 핵심은 특정 행 하나가 아닙니다. **같은 작업을 누가 지금 수행 중인가**를 프로세스 간에 합의해야 합니다.

즉:

- DB 락은 보통 **데이터 중심**
- 분산 락은 보통 **작업 중심**

으로 생각하는 편이 정확합니다.

Phase 2. 많은 경우 DB 락이나 SQL만으로 충분합니다

분산 락을 붙이기 전에 먼저 확인해야 할 것은, 정말로 DB 밖으로 나가야 하는 문제인지입니다.

1. 재고 차감은 보통 Redis 락보다 SQL이 먼저입니다

재고 차감은 자주 이렇게 끝낼 수 있습니다.

```
UPDATE products
SET stock = stock - 1
WHERE id = 1
  AND stock > 0;
```

이 쿼리는:

- 재고가 있을 때만 차감되고
- 품절이면 아무 행도 수정하지 않으며
- 영향받은 행 수로 성공/실패를 판단할 수 있습니다

이 문제를 굳이 Redis 락으로 바꾸면:

- 락 획득
- DB 조회
- DB 갱신
- 락 해제

처럼 단계만 늘어날 수 있습니다. 핵심 규칙이 이미 DB 한 문장으로 표현된다면, **분산 락은 대개 과합니다.**

2. 상태 전이도 조건부 `UPDATE`가 더 직접적일 수 있습니다

```
UPDATE orders
SET status = 'PAID'
WHERE id = 1
  AND status = 'READY';
```

이 경우도:

- `1`행 수정이면 성공
- `0`행 수정이면 이미 다른 상태이거나 선행 조건 불만족

즉, 상태 전이의 핵심이 특정 행의 현재 값이라면 먼저 **조건부 `UPDATE`**를 검토하는 편이 맞습니다.

3. 중복 삽입 방지는 락보다 `UNIQUE` 제약이 더 강합니다

예를 들어 같은 사용자가 같은 쿠폰을 두 번 발급받으면 안 된다면:

```
ALTER TABLE issued_coupon
ADD CONSTRAINT uk_coupon_user UNIQUE (coupon_id, user_id);
```

그 후 그냥 `INSERT`하고 중복 키 예외를 처리하면 됩니다.

이 문제의 핵심은 "동시에 실행되지 않게 하자"가 아니라 **중복 결과를 물리적으로 허용하지 말자**이기 때문입니다.

Phase 3. MySQL `GET_LOCK()` 이면 충분한 경우도 있습니다

분산 락 이야기를 할 때 자주 빠지는 중간 단계가 있습니다. **`named lock`** 또는 **`advisory lock`** 입니다.

MySQL의 `GET_LOCK()` 은 이런 형태입니다.

```
SELECT GET_LOCK('myapp:daily-settlement', 10);
-- 성공하면 1, timeout이면 0

-- 작업 수행

SELECT RELEASE_LOCK('myapp:daily-settlement');
```

이 락은 특정 행이 아니라 **이름(string)** 을 잠급니다. MySQL 문서 기준으로:

- 같은 이름은 다른 세션이 동시에 획득할 수 없고
- 명시적으로 `RELEASE_LOCK()` 하거나 세션이 종료될 때 풀리며
- `COMMIT` 이나 `ROLLBACK` 으로는 풀리지 않습니다

즉, `GET_LOCK()` 은 `row lock` 이 아니라 **협조적** **`advisory lock`** 입니다.

언제 유용할까?

- 같은 MySQL primary 하나를 모든 앱 인스턴스가 공유하고 있고

- 보호 대상이 특정 행이 아니라 "작업 이름"일 때
- 예를 들어 `daily-settlement`, `rebuild-home-cache`, `send-batch-report` 같은 작업을 한 번만 실행하고 싶을 때

이런 경우에는 Redis를 추가로 쓰지 않고도 `named lock`으로 충분할 수 있습니다.

하지만 한계도 분명합니다

MySQL 문서 기준으로 `GET_LOCK()`은 **단일 `mysqld` 기준**입니다. 즉:

- 여러 MySQL 서버 전체에 걸친 범용 분산 락이라고 보기는 어렵고
- 트랜잭션 커밋/롤백과 자동으로 묶이지도 않으며
- 락 이름 충돌을 피하려면 애플리케이션별 네임스페이스를 잘 잡아야 합니다

그래서 문제의 범위가 "우리 앱 인스턴스들 + 하나의 MySQL primary"라면 후보가 될 수 있지만, 더 넓은 분산 조율 문제라면 Redis 같은 외부 락 저장소가 더 자연스럽습니다.

> **참고:** PostgreSQL의 `advisory lock`도 비슷한 목적의 도구입니다. 다만 세부 동작은 DBMS마다 다르므로, 이 글에서는 MySQL `GET_LOCK()` 기준으로만 설명합니다.

Phase 4. Redis 분산 락이 필요한 순간

Redis 락은 보통 **보호 대상이 DB 행이 아닐 때** 의미가 커집니다.

1. 여러 인스턴스에서 같은 작업이 중복 실행되면 안 된다

예를 들어 애플리케이션 서버가 4대이고, 각 서버에서 같은 배치 스케줄러가 기동된다고 가정해 보겠습니다.

```
server-a: 00:00에 정산 배치 시작
server-b: 00:00에 정산 배치 시작
server-c: 00:00에 정산 배치 시작
server-d: 00:00에 정산 배치 시작
```

여기서는 특정 행 하나를 잠그는 것이 아니라 **정산 배치 작업 전체**를 한 번만 실행하고 싶습니다. 이런 경우는 작업 이름 기준의 락이 더 잘 맞습니다.

2. 보호 대상이 외부 자원이다

예를 들어:

- 외부 결제사 정산 API 호출
- 외부 시스템 동기화 작업
- 파일 생성/업로드
- 캐시 재생성

이런 작업은 DB 행 락만으로는 직접 보호할 수 없습니다. DB 트랜잭션 안에서 외부 자원을 잠근다는 개념이 없기 때문입니다.

3. 캐시 재생성처럼 "같은 키 작업"을 한 번만 수행하고 싶다

캐시 스탬피드 글과도 연결되는 부분입니다. 예를 들어 같은 캐시 키가 만료되었을 때 여러 인스턴스가 동시에 재생성을 시작하면 원본 저장소로 요청이 몰릴 수 있습니다.

이때는:

- `cache:rebuild:home-feed`
- `cache:rebuild:item:123`

같은 작업 키를 기준으로 **한 프로세스만 재생성**하게 만들고 싶을 수 있습니다. 이런 경우는 `distributed lock`이 잘 맞는 대표 사례입니다.

Phase 5. Redis 락은 "트랜잭션 락"이 아니라 "`lease`"에 가깝습니다

Redis 공식 문서에서 가장 중요한 포인트 중 하나는 **TTL이 락의 유효 시간이라는 점**입니다.

가장 단순한 획득 패턴은 다음과 같습니다.

```
SET lock:daily-settlement random-token NX PX 30000
```

이 의미는:

- 키가 없을 때만 락을 잡고

- 30초 후에는 자동 만료되며
- 그 30초 안에 작업이 끝난다고 가정한다

는 뜻입니다.

즉, Redis 락은 "작업이 끝날 때까지 영원히 잠근다"가 아니라, **정해진 유효 시간 동안만 배타성을 기대하는 `lease`** 에 가깝습니다.

그래서 TTL을 잘못 잡으면 문제가 생깁니다

예를 들어 작업이 10초 걸릴 줄 알고 TTL을 10초로 잡았는데, 실제로는 GC pause, 네트워크 지연, 외부 API 지연 때문에 25초가 걸릴 수 있습니다. 그러면:

1. 클라이언트 A가 락 획득
2. TTL 만료
3. 클라이언트 B가 같은 락 획득
4. A와 B가 겹쳐서 작업 수행

즉, Redis 락은 **TTL 안에서만 상호 배제를 기대할 수 있다**는 점을 잊으면 안 됩니다.

해제도 `DEL` 만 하면 안 됩니다

Redis 문서는 락 값을 **랜덤 토큰**으로 두고, 해제 시에는 "내가 잡은 락이 아직 맞는지"를 확인한 뒤 지우라고 설명합니다.

```
if redis.call("get",KEYS[1]) == ARGV[1] then
    return redis.call("del",KEYS[1])
else
    return 0
end
```

이 검사가 필요한 이유는 간단합니다.

- A가 락 획득
- TTL 만료
- B가 새 락 획득
- A가 늦게 도착해서 그냥 `DEL` 실행

이렇게 되면 A가 **B의 락을 지워 버릴 수 있기 때문**입니다.

단일 master + replica failover는 완전한 상호 배제를 보장하지 않습니다

Redis 공식 문서는, 단순히 master에 락을 쓰고 replica failover에 기대는 방식은 비동기 복제 때문에 **상호 배제가 깨질 수 있다**고 설명합니다.

예를 들어:

1. 클라이언트 A가 master에 락 획득
2. 복제되기 전에 master 장애
3. replica 승격
4. 클라이언트 B가 같은 락 획득

이 상황에서는 A와 B가 동시에 락을 가진 것처럼 보일 수 있습니다.

Redis 문서는:

- 가끔 이런 race가 나도 괜찮다면 단일 인스턴스 방식이 현실적인 선택일 수 있고
- 그렇지 않다면 독립 master 다수에 기반한 **`Redlock`** 같은 더 강한 알고리즘을 검토하라고 설명합니다

> **참고:** 여기서 중요한 것은 "무조건 `Redlock`을 써야 한다"가 아니라, **내가 허용할 수 있는 실패 모델이 무엇인가**를 먼저 정하는 것입니다.

Phase 6. 결국 어떻게 고르면 될까?

질문을 바꾸면 판단이 빨라집니다.

질문 1. 보호 대상이 특정 DB 행인가, 작업 이름인가?

- 특정 행이다 → `row lock`, 조건부 `UPDATE`, `UNIQUE`를 먼저 봅니다
- 작업 이름이다 → `advisory lock`이나 `distributed lock` 후보입니다

질문 2. 모든 인스턴스가 공유하는 단일 DB가 이미 있는가?

- 있다 → `GET_LOCK()` 같은 `named lock`도 후보가 됩니다
- 없다 / 더 넓은 분산 조율이 필요하다 → Redis 락이 더 자연스럽습니다

질문 3. 락 없이 더 강한 정합성을 만들 수 있는가?

- 중복 삽입 → `UNIQUE`
- 조건부 상태 변경 → 조건부 `UPDATE`
- 중복 API 요청 → `idempotency key`

이런 식으로 해결되면 락보다 더 단순하고 명확합니다.

질문 4. Redis 락의 실패 모델을 감당할 수 있는가?

- TTL 만료 전 작업 완료를 reasonably 보장할 수 있는가?
- `owner token` 검증으로 안전하게 해제하고 있는가?
- failover 중 드문 중복 실행을 허용할 수 있는가?

이 질문에 답이 모호하면, "Redis 락을 붙이면 끝"이 아닐 가능성이 큽니다.

한눈에 보는 선택 기준

문제 유형	더 먼저 볼 수단	이유
재고 차감, 상태 전이, 같은 행 수정 경쟁	`FOR UPDATE`, 조건부 `UPDATE`	보호 대상이 DB 행입니다
중복 삽입 방지	`UNIQUE` 제약	락보다 더 직접적으로 막을 수 있습니다
단일 MySQL primary 기준의 작업 단일 실행	`GET_LOCK()`	작업 이름 기준 `advisory lock`이면 충분할 수 있습니다
여러 인스턴스의 외부 작업 조율	Redis 락	특정 행이 아니라 작업 키를 보호해야 합니다
캐시 재생성, 스케줄러 중복 실행	Redis 락 또는 `named lock`	같은 작업을 한 번만 수행하고 싶습니다
아주 강한 상호 배제가 필요한 분산 환경	단순 Redis lock 이상 검토	TTL, failover, 알고리즘 가정을 함께 봐야 합니다

정리

1. **DB 락과 분산 락은 푸는 문제가 다릅니다** — 하나는 데이터 변경 경쟁을, 다른 하나는 여러 프로세스의 실행 경쟁을 주로 다룹니다
2. **많은 동시성 문제는 DB 안에서 끝납니다** — 조건부 `UPDATE`, `UNIQUE`, `FOR UPDATE`가 먼저인 경우가 많습니다

3. **MySQL `GET_LOCK()` 같은 `advisory lock`은 Redis 이전에 볼 수 있는 중간 선택지입니다** — 특히 단일 DB를 공유하는 다중 인스턴스 환경에서 유용할 수 있습니다
4. **Redis 락은 외부 자원이나 작업 키를 조율할 때 빛납니다** — 스케줄러 단일 실행, 캐시 재생성, 외부 작업 직렬화가 대표적입니다
5. **Redis 락은 `lease` 모델이라는 점을 잊으면 안 됩니다** — TTL, `owner token` 검증, failover 시나리오를 함께 봐야 합니다

핵심을 한 문장으로 줄이면 이렇습니다.

> **"같은 데이터를 누가 바꾸는가"의 문제는 DB에서, "같은 작업을 누가 실행하는가"의 문제는 `named lock`이나 `distributed lock`에서 보는 편이 보통 더 정확합니다**

Chapter 10

멱등성 완전 정복 — 중복 요청을 한 번처럼 처리하는 법

#동시성 제어

HTTP 메서드의 멱등성과 `idempotency key` 기반 API 멱등성을 구분하고, 중복 결제·중복 주문을 저장과 재시도 관점에서 어떻게 제어할지 정리합니다.

멱등성, 왜 알아야 하나요?

낙관적 락 vs 비관적 락 글과 분산 락은 언제 써야 할까 글까지 읽고 나면 이런 질문이 남습니다.

- 결제 요청이 timeout 나서 클라이언트가 다시 보내면, 같은 결제가 두 번 실행되지 않게 하려면 어떻게 해야 할까요?
- 주문 생성 API에 분산 락을 걸기보다 `idempotency key`로 끝낼 수 있는 경우는 언제일까요?
- HTTP에서 `PUT`이 멱등하다는 말과, 결제 API에서 말하는 멱등성은 같은 뜻일까요?
- `UNIQUE` 제약 조건, 락, 멱등성은 각각 무엇을 막는 도구일까요?

핵심은 이것입니다. **멱등성은 "동시에 실행되지 않게 막는 기술"이 아니라, 같은 요청이 여러 번 와도 결과를 한 번처럼 보이게 만드는 계약**입니다.

즉, 락이 충돌을 줄 세우는 도구라면, 멱등성은 **재시도와 중복 요청을 흡수하는 도구**에 가깝습니다.

이 글은 HTTP semantics와 일반적인 API 서버 설계를 기준으로, **멱등성이 무엇을 보장하고 무엇은 보장하지 않는지**, 그리고 `idempotency key`를 실제로 어떻게 저장하고 해석해야 하는지 정리합니다.

먼저 선택 기준부터 보면

실무에서는 보통 아래 순서로 판단하면 덜 헷갈립니다.

상황	먼저 볼 선택지	이유
같은 DB 행 수정 경쟁	조건부 `UPDATE`, `FOR UPDATE`	핵심 문제는 동시 수정입니다
중복 삽입 방지	`UNIQUE` 제약 조건	결과 자체를 물리적으로 금지할 수 있습니다
네트워크 재시도, 중복 클릭, timeout 후 재요청	`idempotency key`	같은 요청을 한 번처럼 처리해야 합니다
외부 API 호출의 중복 실행 방지	멱등성 + 필요 시 락	중복 요청 흡수와 직렬화는 다른 문제입니다
배치/스케줄러 단일 실행	`named lock`, `distributed lock`	작업 중복 실행 조율이 핵심입니다

가장 짧게 줄이면 이렇습니다.

- **동시 수정 문제는 락이나 SQL로 봅니다**

- **재시도와 중복 요청 문제는 멱등성으로 봅니다**
- **중복 결과 자체를 금지해야 하면 `UNIQUE` 가 더 직접적입니다**

Phase 1. HTTP의 멱등성과 API 설계의 멱등성은 다릅니다

여기서 가장 자주 헷갈립니다.

HTTP 메서드의 멱등성

RFC 9110 기준으로 `PUT`, `DELETE` 같은 `idempotent` 메서드는 같은 요청을 반복해도 의도된 최종 상태가 같아야 합니다. `GET`, `HEAD`, `OPTIONS`, `TRACE` 처럼 `safe` 한 메서드도 이 성질을 가집니다.

의미는 단순합니다.

> **같은 요청을 여러 번 보내더라도, 의도된 최종 서버 상태는 한 번 보낸 것과 같아야 합니다**

예를 들어:

```
PUT /users/1
DELETE /posts/10
```

같은 `PUT` 을 두 번 보내도 최종 리소스 상태가 같다면 멱등합니다. 같은 `DELETE` 를 여러 번 보내도 "결국 삭제된 상태"라면 멱등합니다.

중요한 점은 **응답이 항상 같아야 한다는 뜻이 아니라, 의도된 효과가 같아야 한다는 뜻**입니다.

애플리케이션 레벨 멱등성

반면 결제 API나 주문 생성 API는 보통 `POST`를 사용합니다.

```
POST /payments
POST /orders
```

이런 요청은 HTTP 메서드 자체로는 멱등하지 않습니다. 하지만 애플리케이션이 `idempotency key`를 도입하면:

- 클라이언트가 같은 요청을 다시 보내더라도
- 서버가 같은 작업으로 인식하고
- 새 결제/새 주문을 다시 만들지 않게 할 수 있습니다

즉:

- **HTTP 멱등성** — 메서드 semantics
- **애플리케이션 멱등성** — 같은 비즈니스 요청을 한 번처럼 처리하는 서버 계약

으로 구분하는 편이 정확합니다.

Phase 2. `idempotency key`는 무엇을 보장할까?

`idempotency key`의 역할은 대개 다음 한 문장으로 정리됩니다.

> **"이 요청은 같은 작업의 재시도입니다"라는 사실을 서버에 알려 주는 키**

예를 들어 결제 요청에서:

```
POST /payments
Idempotency-Key: pay:order-123
```

서버는 이 키를 보고 판단합니다.

- 처음 보는 키인가? → 새 작업으로 처리
- 이미 처리한 키인가? → 이전 결과를 재사용하거나 같은 작업 결과로 응답

핵심은 **클라이언트 재시도를 서버가 중복 실행으로 오해하지 않게 만드는 것**입니다.

어떤 상황에서 특히 필요할까?

- 결제 버튼을 사용자가 두 번 클릭
- 모바일 네트워크 불안정으로 응답을 못 받고 재요청
- 서버는 처리했지만 클라이언트는 timeout으로 실패로 인식
- API gateway나 client library가 자동 재시도 수행

이런 상황에서 멱등성이 없으면:

- 결제가 두 번 승인되고

- 주문이 두 건 생성되고
- 쿠폰이 두 번 발급되고
- 같은 외부 요청이 반복 실행될 수 있습니다

무엇을 보장하지는 않을까?

`idempotency key`는 만능이 아닙니다.

- 동시 수정 경쟁 자체를 자동으로 막아 주지는 않습니다
- 잘못된 비즈니스 로직을 자동으로 바로잡지 않습니다
- key를 늦게 저장하거나, side effect를 먼저 발생시키면 중복 실행을 막지 못할 수 있습니다

즉, 멱등성은 **중복 요청 흡수 계약**이지, 모든 정합성 문제의 대체재는 아닙니다.

Phase 3. 보통은 이렇게 저장합니다

실무에서는 `idempotency_key` 전용 저장소를 따로 둡니다.

가장 단순한 예시는 이런 테이블입니다.

```
CREATE TABLE api_idempotency (
  idempotency_key VARCHAR(255) PRIMARY KEY,
  request_hash    VARCHAR(64)  NOT NULL,
  status          VARCHAR(20)  NOT NULL,
  response_code   INT          NULL,
  response_body   JSON         NULL,
  resource_type   VARCHAR(50)  NULL,
  resource_id      VARCHAR(100) NULL,
  created_at      TIMESTAMP    NOT NULL,
  updated_at      TIMESTAMP    NOT NULL
);
```

핵심 컬럼은 보통 이 정도입니다.

- `idempotency_key` — 같은 작업인지 식별
- `request_hash` — 같은 key에 다른 payload가 들어오는 오용 방지
- `status` — `PROCESSING`, `SUCCEEDED`, `FAILED`
- `response_code`, `response_body` — 재시도 시 같은 응답 재사용
- `resource_type`, `resource_id` — 실제 생성된 리소스 추적

왜 `request_hash`가 필요할까?

같은 key를 재사용했는데 payload가 다르면, 서버는 난감해집니다.

```
요청 A: Idempotency-Key = pay:123, amount = 10000
요청 B: Idempotency-Key = pay:123, amount = 50000
```

이 둘을 같은 작업으로 보면 잘못이고, 다른 작업으로 보면 key 의미가 깨집니다.

그래서 많은 시스템은:

- 같은 key + 같은 payload → 재시도
- 같은 key + 다른 payload → 오류

로 처리합니다.

Stripe 문서도 같은 key를 재사용하면서 파라미터가 다르면 오류를 반환하는 방향을 설명합니다.

Phase 4. 보통의 처리 흐름은 이렇습니다

예를 들어 결제 생성 API를 생각해 보겠습니다.

```
1. 클라이언트가 `Idempotency-Key`와 함께 요청 전송
2. 서버가 key 존재 여부 확인
3. 없으면 `PROCESSING` 상태로 먼저 기록
4. 실제 비즈니스 로직 수행
5. 성공/실패 결과를 key 레코드에 저장
6. 이후 같은 key 요청이 오면 저장된 결과 재사용
```

이를 아주 단순화하면 이런 흐름입니다.

```
fun createPayment(command: CreatePaymentCommand, key:
String): PaymentResponse {
    val existing = idempotencyRepository.find(key)
    if (existing ≠ null) {
        return existing.toResponse()
    }

    idempotencyRepository.insertProcessing(key,
hash(command))

    val payment = paymentService.create(command)
    val response = PaymentResponse.from(payment)

    idempotencyRepository.markSucceeded(key, response)
    return response
}
```

물론 실제 구현에서는 여기서 더 조심해야 합니다.

1. "존재 확인 후 insert"만 하면 race가 날 수 있습니다

두 요청이 동시에 들어오면:

```
A: key 없음 확인
B: key 없음 확인
A: insert
B: insert
```

가 될 수 있습니다.

그래서 보통은:

- `idempotency_key`에 `PRIMARY KEY` 또는 `UNIQUE`를 두고
- `INSERT` 자체를 원자적으로 시도한 뒤
- 중복 키 충돌 시 기존 레코드를 읽는 방식

으로 갑니다.

즉, 멱등성 저장소 자체도 **DB 제약 조건 위에** 세우는 편이 안전합니다.

2. `PROCESSING` 상태가 왜 필요할까?

첫 요청이 아직 끝나지 않았는데 같은 key가 다시 들어올 수 있습니다.

예를 들어:

1. 요청 A 시작
2. `idempotency_key` 저장
3. 외부 결제사 호출 중 timeout
4. 클라이언트 재시도

이때 같은 key가 이미 `PROCESSING`이면 서버는 선택해야 합니다.

- 아직 처리 중이라고 `409 Conflict`나 `202 Accepted` 계열로 안내
- 짧게 poll 하거나 대기 후 최종 결과 반환
- 내부 정책상 같은 작업의 완료를 기다림

핵심은 **"아직 끝나지 않은 같은 작업"** 도 별도 상태로 취급해야 한다는 점입니다.

3. 성공만 저장할지, 실패도 저장할지 정책이 필요합니다

여기서 자주 헷갈립니다.

- validation error처럼 "아예 실행이 시작되지 않은 실패"
- 외부 호출 후 일부 작업이 끝난 뒤 난 실패
- 확정적인 비즈니스 실패와 일시적 인프라 실패

는 성격이 다릅니다.

Stripe 문서는 "실행이 시작된 뒤의 첫 결과"를 저장하고 재사용하는 방향을 설명합니다. 하지만 여러분의 API는:

- 확정적 실패는 저장하고
- 재시도 가능한 일시 오류는 저장하지 않거나
- 별도 상태로 두고 다시 시도 가능하게

설계할 수도 있습니다.

즉, **멱등성 저장 전략은 실패 모델과 함께 설계해야 합니다.**

Phase 5. 결제, 주문, 쿠폰에서는 어떻게 다를까?

1. 결제 생성

결제는 멱등성이 가장 자주 필요한 예시입니다.

```
POST /payments
Idempotency-Key: pay:order-123
```

클라이언트가 timeout 후 재시도하더라도:

- 같은 `payment`를 재생성하지 않고
- 같은 결제 결과를 돌려주거나
- 이미 생성된 결제 리소스를 가리켜야 합니다

결제는 외부 시스템까지 얽히는 경우가 많기 때문에, **락보다 멱등성이 먼저**인 경우가 많습니다.

2. 주문 생성

주문도 비슷합니다.

- 장바구니에서 주문 생성 버튼을 두 번 누름
- 프론트엔드가 재시도
- API gateway가 재전송

이 상황에서 주문이 두 건 만들어지면 안 됩니다.

이때는 `order:create:{cartId}` 같은 key를 둘 수 있습니다. 다만 여기서 중요한 것은:

- 같은 cart에서 여러 주문이 정말 불가능한가?
- key 스코프를 user 기준으로 잡을지, cart 기준으로 잡을지

를 먼저 정해야 한다는 점입니다.

3. 쿠폰 발급

쿠폰 발급은 멱등성만으로 끝나지 않는 경우가 많습니다.

예를 들어 "사용자당 1회 발급"이라면:

- 중복 요청 흡수는 `idempotency key`
- 결과 자체의 중복 방지는 `UNIQUE (coupon_id, user_id)`

처럼 두 층이 함께 필요할 수 있습니다.

즉:

- 멱등성은 **같은 요청의 재시도**를 다루고
- `UNIQUE`는 **비즈니스 결과의 중복 자체**를 막습니다

Phase 6. 락, `UNIQUE`, 멱등성은 역할이 다릅니다

이 셋은 대체 관계라기보다 서로 다른 층에서 문제를 푸는 도구입니다.

락

- 같은 자원을 동시에 수정하지 않게 조율
- 읽기-판단-쓰기 경쟁 제어
- 예: 재고 차감, 상태 전이

`UNIQUE`

- 중복 결과 자체를 DB에서 금지
- 예: 같은 쿠폰의 중복 발급, 같은 외부 주문 ID의 중복 저장

멱등성

- 같은 요청의 재시도와 중복 제출 흡수
- 예: timeout 후 결제 재요청, 버튼 연타, 네트워크 재전송

예를 들어 결제 API는 이렇게 조합될 수 있습니다.

- 클라이언트 재시도 흡수 → `idempotency key`
- DB 중복 저장 방지 → `UNIQUE (external_payment_id)`
- 특정 상태 전이 보호 → 조건부 `UPDATE` 또는 락

즉, **멱등성이 락을 없애는 것이 아니라, 다른 종류의 문제를 앞단에서 흡수하는 것**입니다.

Phase 7. 자주 하는 실수

1. key만 받고 서버에 저장하지 않는다

클라이언트가 `Idempotency-Key`를 보내더라도, 서버가 그 key와 결과를 저장하지 않으면 아무 의미가 없습니다.

2. key 스코프를 너무 넓거나 너무 좁게 잡는다

예를 들어:

- 너무 넓음: `user:123`
- 너무 좁음: 매 요청마다 무작위 key 생성

이 둘 다 문제입니다. key는 **"같은 작업"을 안정적으로 식별할 정도로만** 잡아야 합니다.

3. payload가 다른데 같은 key를 허용한다

이 경우는 재시도와 새 요청이 구분되지 않습니다. 같은 key에 대해 입력이 달라지면 보통 오류로 처리하는 편이 낫습니다.

4. side effect보다 늦게 key를 저장한다

외부 결제 호출을 먼저 하고, 그 다음 key를 저장하면 이미 중복 실행을 막을 기회를 놓친 것입니다.

5. "멱등성 = 정확히 한 번"이라고 생각한다

멱등성은 보통 **중복 요청을 한 번처럼 처리하는 계약**이지, 분산 시스템 전체에서 철저한 `exactly once`를 공짜로 보장하는 마법은 아닙니다.

특히 외부 시스템과 webhooks까지 얽히면:

- 요청 측 멱등성
- 저장 측 `UNIQUE`

- 소비 측 중복 이벤트 처리

를 함께 설계해야 합니다.

한눈에 보는 선택 기준

문제 유형	더 먼저 볼 수단	이유
같은 행 수정 경쟁	조건부 `UPDATE`, `FOR UPDATE`	핵심은 동시 수정입니다
중복 결과 자체 금지	`UNIQUE` 제약	결과를 DB에서 직접 막습니다
timeout 후 재요청, 버튼 연타	`idempotency key`	같은 요청 재시도를 흡수해야 합니다
외부 API 중복 호출 방지	멱등성 우선, 필요 시 락 보조	재시도 흡수와 직렬화는 다른 문제입니다
스케줄러 단일 실행	`named lock`, `distributed lock`	작업 중복 실행 조율이 핵심입니다

정리

1. **HTTP 메서드의 멱등성과 API 설계의 멱등성은 다릅니다** — 하나는 HTTP semantics이고, 다른 하나는 같은 비즈니스 요청을 한 번처럼 처리하는 서버 계약입니다

2. **`idempotency key`는 중복 요청 흡수 도구입니다** — timeout, 재시도, 버튼 연타 같은 상황에서 특히 중요합니다
3. **멱등성 저장소는 보통 key, payload hash, 처리 상태, 응답을 함께 관리합니다** — 같은 key 재사용과 `PROCESSING` 상태를 구분해야 합니다
4. **멱등성만으로 모든 정합성 문제가 끝나지는 않습니다** — `UNIQUE`, 조건부 `UPDATE`, 락과 각자 역할이 다릅니다
5. **가장 흔한 실수는 key를 너무 늦게 저장하거나, 같은 key에 다른 payload를 허용하는 것입니다**

핵심을 한 문장으로 줄이면 이렇습니다.

> **락이 충돌을 줄 세우는 도구라면, 멱등성은 재시도와 중복 요청을 한 번처럼 흡수하는 도구입니다**