5. 논리적 모델링1 - 키

#0.강의/2.데이터베이스로드맵/3.설계1

다양한 종류의 키

데이터베이스 설계의 3단계

우리는 앞서 개념적 모델링을 통해 '쇼핑몰'의 청사진인 ERD를 완성했다. 이제부터는 이 청사진을 실제 관계형 데이터베이스에 가깝게 다듬는 **논리적 모델링** 단계로 들어간다. 그러기 위해서 지금부터 논리적 모델링을 배워보자.

그전에 먼저 데이터베이스 설계의 세 가지 단계를 다시 한번 정리해보자.

1. **개념적 모델링 (Conceptual Modeling)**: 개념적 모델링은 복잡한 현실 세계의 비즈니스 요구사항을 누구나 이해하기 쉬운 그림으로 단순화하는 것에 초점을 맞춘다. "어떤 데이터가 필요한가?", "데이터들은 서로 어떤 관계를 맺고 있는가?"와 같은 근본적인 질문에 답하는 단계라고 할 수 있다.
   이 단계는 비즈니스 요구사항을 이해하고, 데이터의 큰 그림을 그리는 과정이다. 어떤 데이터가 필요한지(엔티티), 데이터들 사이에 어떤 관계가 있는지(관계)를 식별하여 **ERD(Entity-Relationship Diagram)** 로 표현한다. 우리가 이전 시간에 '회원', '상품', '주문' 같은 엔티티를 도출하고 그 관계를 정의한 것이 바로 개념적 모델링이다. 이것은 마치 건물의 아이디어를 스케치하는 것과 같다.

2. **논리적 모델링 (Logical Modeling)**: 논리적 모델링은 개념적 모델링에서 만든 청사진을 **관계형 데이터베이스 구조**에 맞게 변환하는 과정이다. 엔티티는 **테이블**로, 속성은 **컬럼**으로 바꾸고, 각 테이블의 **기본 키(Primary Key)** 와 테이블 간의 관계를 표현하는 **외래 키(Foreign Key)** 등을 정의한다. 이 단계는 특정 관계형 데이터베이스 시스템(MySQL, Oracle 등)에 종속되지 않는, 순수한 관계형 데이터베이스 모델을 만드는 것이 목표다. 건물의 기본 설계도를 그리는 단계라고 할 수 있다.

> 릴레이션 용어
> 논리적 모델링 단계에서 학술적으로는 **릴레이션(Relation)**, **튜플(Tuple)**, **속성(Attribute)**이라는 용어를 사용한다. 하지만 실무에서는 거의 쓰이지 않고, 대신에 **테이블(Table)**, **행(Row)**, **컬럼(Column)**이라는 용어를 사용한다. 본 강의에서도 이해하기 쉽게 '테이블'이라는 용어를 사용하겠다.

3. **물리적 모델링 (Physical Modeling)**: 논리적 모델을 실제 사용할 데이터베이스 시스템(우리의 경우 MySQL)에 최적화하여 구현하는 마지막 단계다. 각 컬럼의 구체적인 데이터 타입(예: `VARCHAR(50)`, `BIGINT`)을 결정하고, 성능 최적화를 위한 인덱스를 설정하며, 파일 저장 방식 등 물리적인 요소를 고려한다. 건물의 시공을 위해 철근의 종류나 콘크리트의 강도까지 정하는 상세 시공 설계도를 만드는 것과 같다.

논리적 모델링은 개념적 모델을 **관계형 데이터베이스 이론에 맞게** 변환하는 과정이다. 특정 RDBMS(MySQL, Oracle 등)에 종속되지 않는, 순수한 데이터 구조를 설계한다. 이 과정에서 가장 먼저, 그리고 가장 중요하게 다루어야 할 주제가 바로 **'키(Key)'**다.

왜 '키'가 그렇게 중요한가?

상상해보자. 우리 쇼핑몰에 '션'이라는 이름을 가진 회원이 수십 명이라면, 특정 '션' 회원의 주문 내역을 어떻게 정확히 찾아낼 수 있을까? 만약 상품 테이블에 이름과 가격이 똑같은 상품이 두 개 등록되어 있다면, 고객이 주문한 상품이 둘 중 어느 것인지 어떻게 구별할까?

이러한 문제들을 해결하기 위해 데이터베이스는 각 데이터 행(Row)을 **유일하게 식별할 수 있는 장치**가 필요하다. 그것이 바로 **키(Key)**다.

키는 단순히 데이터를 찾는 용도로만 쓰이지 않는다. 테이블과 테이블을 연결하여 관계를 맺어주고(예: 어떤 회원이 어떤 주문을 했는지), 데이터가 중복되거나 잘못 입력되는 것을 막아주는 **무결성 제약조건**의 역할도 수행한다. 즉, 키는 데이터베이스 설계의 가장 근본적인 주춧돌이며, 데이터의 신뢰성을 보장하는 핵심 장치다.

모든 데이터를 구별하는 열쇠, 키(Key)

키(Key)는 테이블에 있는 각각의 행(Row)을 고유하게 식별할 수 있는 하나 이상의 컬럼 집합이다. 키의 종류는 여러 가지가 있으며, 각각의 역할과 의미를 명확하게 이해해야 한다.

기본 키 (Primary Key - PK)

- 테이블의 모든 행을 유일하게 식별하는 **대표 키**다.
- 기본 키가 반드시 지켜야 하는 3가지 규칙

1. **NULL 값을 가질 수 없다** (`NOT NULL`)
2. **반드시 유일해야 한다**. (`UNIQUE`)
3. **값이 변하지 않아야 한다 (불변성)** - 이론적으로 기본 키의 값을 변경할 수는 있다. 하지만 기본 키 값이 변하면 이 키를 참조하는 모든 외래 키(FK)와 데이터 무결성에 큰 문제가 생긴다. **실무에서는 절대로 기본 키의 값을 변경하면 안된다.**

- 예: `회원` 테이블의 `member_id`, `상품` 테이블의 `product_id`가 바로 기본 키다. 이 키만 있으면 우리는 수백만 명의 회원 중에서 정확히 한 명을, 수십만 개의 상품 중에서 정확히 하나를 콕 집어낼 수 있다.

후보 키 (Candidate Key)

- 후보 키는 말 그대로 기본 키가 될 수 있는 '후보 선수들'이다.
- 기본 키가 될 수 있는 자격, 즉 **유일성**과 **최소성**을 모두 만족하는 모든 키를 말한다.
  - **유일성(Unique)**: 모든 행을 서로 구분할 수 있어야 한다.
  - **최소성(Minimality)**: 행을 유일하게 식별하는 데 꼭 필요한 최소한의 컬럼만 포함해야 한다. 뒤에서 자세히 설명한다.
- 예: `회원` 테이블에서 `member_id` 외에 `이메일(email)`이나 `연락처(phone_number)`도 모든 회원이 서로 다른 값을 가진다고 규칙을 정한다면 (UNIQUE 제약조건), 이 컬럼들도 후보 키가 될 수 있다.

대체 키 (Alternate Key)

- 후보 키 중에서 **기본 키로 선택되지 않은 나머지 키**들을 말한다.
- 만약 `member_id`를 기본 키로 선택했다면, 후보 키였던 `이메일`과 `연락처`는 대체 키가 된다. 이들 역시 데이터를 찾는 데 유용하게 사용할 수 있다.

외래 키 (Foreign Key - FK)

- 가장 중요한 키 중 하나로, **테이블 간의 관계를 연결하는 역할**을 한다.
- 한 테이블의 컬럼이 다른 테이블의 **기본 키**를 참조하는 것이다.
- '쇼핑몰'의 `orders` 테이블에 있는 `member_id` 컬럼을 생각해보자. 이 컬럼에는 주문한 회원의 ID가 저장된다. 이 값은 `member` 테이블에 있는 `member_id`(기본 키) 값 중 하나여야만 한다. 이렇게 `orders` 테이블의 `member_id`가 `member` 테이블을 참조할 때, 이 컬럼을 외래 키라고 부른다.
- 외래 키 덕분에 우리는 주문 정보만 보고도 어떤 회원이 주문했는지 정확히 알 수 있으며, 이 외래 키 컬럼에 외래 키 제약조건을 걸어주면 존재하지 않는 회원이 주문하는 것과 같은 데이터 불일치를 원천적으로 막을 수 있다.

최소성이란?

유일성을 만족하더라도, 불필요한 컬럼을 포함하여 최소성을 만족하지 못하는 키는 후보 키가 될 수 없다.
참고로 키를 구성할 때 여러 컬럼을 묶어서 하나의 키를 구성할 수 있는데, 이것을 **복합키**라고 한다.

최소성 예시1: 회원

예를 들어 다음과 같은 테이블과 컬럼이 있다고 가정하자.

```
회원(member_id, email, phone_number, name)
```

- `{member_id, email}` → 모든 회원을 구분할 수 있지만, `member_id` 하나만으로도 구분 가능하므로 **최소성을 위배**한다. 따라서 후보 키가 될 수 없다.
- `{email, phone_number}` → 모든 회원을 구분 가능하지만, `email` 하나만으로도 구분 가능하므로 **최소성을 위배**한다. 따라서 후보 키가 될 수 없다.
- `member_id` → 유일하고 최소 → 후보 키 가능
- `email` → 유일하고 최소 → 후보 키 가능
- `phone_number` → 유일하고 최소 → 후보 키 가능

최소성 예시2: 강의실 좌석

강의실의 좌석 배치를 생각해보자. 각 좌석을 구분하기 위한 테이블 `classroom_seat`가 있다고 가정한다.
이 테이블은 학생들의 좌석을 **행(row)**과 **열(column)**로 관리한다.

[classroom_seat 테이블 예시]

row	col	student_name
1	1	김철수
1	2	이영희
2	1	박지성
2	2	손흥민

이 테이블에서 특정 좌석을 유일하게 찾아내기 위한 후보 키로 `{row, col}` 복합키를 생각해 볼 수 있다.

1. **유일성 확인**
   `{row, col}`의 조합은 전체 테이블에서 항상 유일한가? 그렇다. `{1, 1}`이라는 조합은 '김철수' 학생의 자리 하나만을 특정한다. 다른 좌석이 `{1, 1}` 값을 가질 수 없으므로 **유일성을 만족**한다.

2. **최소성 확인**
   그렇다면 이 조합에서 어느 하나라도 빼도 유일할까?

   - `row`만으로 좌석을 식별할 수 있는가? **아니다.** `row`가 1인 좌석은 '김철수'와 '이영희' 두 명이 있다.
   - `col`만으로 좌석을 식별할 수 있는가? **아니다.** `col`이 1인 좌석은 '김철수'와 '박지성' 두 명이 있다.

결론적으로 `{row, col}` 조합은 **유일성**을 만족하고, 두 컬럼 중 어느 하나라도 빠지면 유일성이 깨지므로 **최소성** 또한 만족한다. 따라서 `{row, col}`은 최소성을 만족하는 복합키이자 후보 키다.

자연 키 vs 대리 키1 - 자연 키

모든 테이블에는 각 행(row)을 유일하게 식별할 수 있는 **기본 키(Primary Key, PK)**가 반드시 필요하다. 이 기본 키를 선택하는 과정에서 우리는 두 가지 후보, 즉 '**자연 키(Natural Key)**'와 '**대리 키(Surrogate Key)**'를 만나게 된다. 이 선택은 단순히 키 하나를 정하는 것을 넘어, 미래에 만들어질 시스템 전체의 유연성과 안정성을 결정하는 매우 중요한 설계 결정이다.

자연 키(Natural Key)란 무엇인가?

자연 키란 이름 그대로, 우리의 비즈니스 로직 안에서 **자연스럽게 발생하는, 의미를 가진 데이터**를 기본 키로 사용하는 것을 말한다. 예를 들어, `대한민국 국민`이라는 데이터가 있다면 '주민등록번호'가 자연 키가 될 수 있고, `도서` 데이터에서는 'ISBN' 코드가 자연 키가 될 수 있다.

장점

- **직관성**: 키 값(`test@example.com`, `19900101-000000` 등) 자체에 의미가 담겨 있어, 이 값만 봐도 어떤 데이터를 가리키는지 쉽게 이해할 수 있다.
- **중복 방지**: 비즈니스 규칙상 고유해야 하는 값을 PK로 지정하므로, 데이터베이스 차원에서 값의 중복을 원천적으로 막을 수 있다. (PK는 `NOT NULL` + `UNIQUE` 제약조건)

자연 키의 치명적 약점: '변경'이라는 시한폭탄

자연 키의 장점은 명확하지만, 실무에서 사용할 때 자연 키는 치명적인 약점을 가지고 있다. 그것은 바로 **'변경 가능성'**이다.

현대 데이터베이스 설계의 가장 중요한 원칙 중 하나는 **"기본 키는 영원히 변하지 않아야 한다(Immutable)"**는 것이다. 하지만 비즈니스 로직에 종속되는 자연 키는 이 원칙을 지키기가 매우 어렵다.

기본 키의 값은 이론적으로 변경할 수 있다. 실제 데이터베이스에서 변경도 가능하다.
하지만 실무에서 기본 키의 값은 절대로 변경하면 안된다. 그럼 왜 기본 키의 값을 변경하면 안되는지 예시로 확인해보

자.

시나리오: 회원의 이메일을 PK로 사용한 쇼핑몰의 재앙

우리 쇼핑몰이 처음 문을 열 때, "이메일은 로그인 ID이며 절대 변경할 수 없다"는 정책을 세우고 `member` 테이블의 PK를 `email`로 정했다고 상상해 보자.

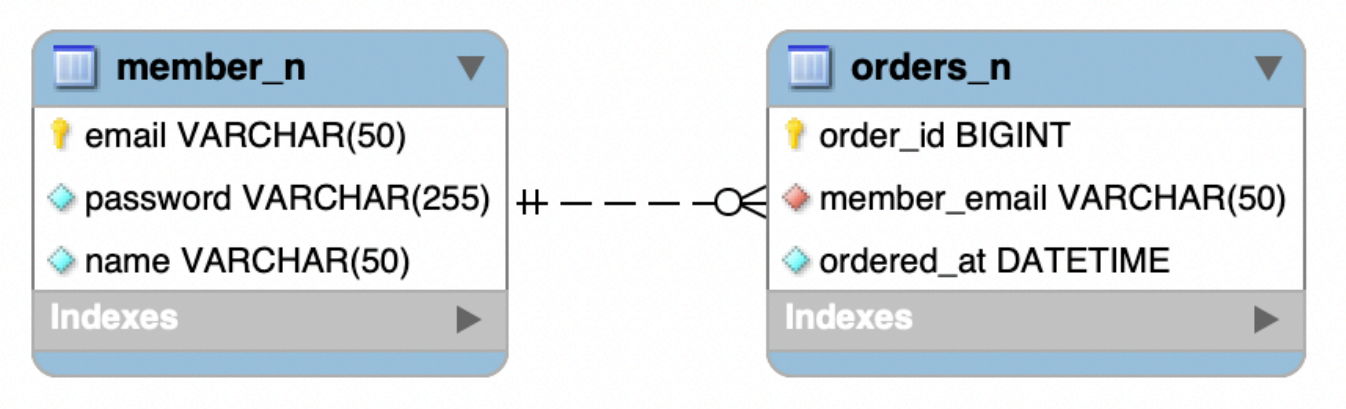
실습 준비

```
-- 데이터베이스가 존재하지 않으면 생성
CREATE DATABASE IF NOT EXISTS my_shop3;
USE my_shop3;
```

- 실습에서는 `my_shop3`이라는 데이터베이스를 사용하겠다.

```
DROP TABLE IF EXISTS orders_n;
DROP TABLE IF EXISTS member_n;

-- 잘못된 설계의 시작
CREATE TABLE member_n (
    email       VARCHAR(50)  NOT NULL, -- PK (자연 키)
    password    VARCHAR(255) NOT NULL,
    name        VARCHAR(50)  NOT NULL,
    PRIMARY KEY (email)
);

-- member 테이블을 참조하는 orders 테이블
CREATE TABLE orders_n (
    order_id      BIGINT       NOT NULL AUTO_INCREMENT,
    member_email  VARCHAR(50)  NOT NULL, -- FK
    ordered_at    DATETIME     NOT NULL,
    PRIMARY KEY (order_id),
    CONSTRAINT fk_orders_member_n FOREIGN KEY (member_email)
        REFERENCES member_n (email)
);
```

- 자연 키(Natural Key)를 사용하는 예시이므로 학습 목적으로 테이블을 쉽게 구분하기 위해 뒤에 `_n`을 붙였다.



- ERD 보는 방법
  - 열쇠: PK
  - 붉은색 마름모: FK
  - 파란색 마름모: 컬럼
  - 꽉 찬 마름모: NOT NULL
  - 빈 마름모: NULL 허용

몇 달 후, 고객 지원팀에서 요청이 온다. "수 많은 회원들이 이메일을 변경하고 싶어 해요. 기능을 만들어주세요."
회사에서 긴 논의 끝에 '이메일은 로그인 ID이며 절대 변경할 수 없다'는 정책을 변경해서 이메일을 변경할 수 있게 허용한다.

이제 모든 것이 꼬이기 시작한다.

회원 `member1@old.com`이 이메일을 `member1@new.com`으로 변경하려고 한다. `member` 테이블의 PK를 직접 변경하는 `UPDATE` 쿼리를 실행해야 한다.

```
-- 실행하는 순간 재앙이 시작될 수 있는 쿼리
UPDATE member_n SET email = 'member1@new.com' WHERE email = 'member1@old.com';
```

기본 키의 값을 변경하는 이 쿼리를 실행하면 어떤 일이 벌어질까? 시스템의 신뢰도와 안정성 자체를 파괴할 수 있는 다양한 문제들이 발생한다.

문제 1: 참조 무결성 제약조건 위반
가장 먼저 마주하는 현실적인 문제는, 앞서 실행한 `UPDATE` 쿼리가 애초에 **성공하지 않는다**는 점이다. 왜일까? 바로 데이터베이스가 스스로를 보호하기 위해 설정해 둔 **외래 키(Foreign Key) 제약조건** 때문이다.

우리는 `orders_n` 테이블을 생성할 때 외래 키 제약조건을 설정했다. 이는 '자식 테이블(`orders_n`)에서 참조하고

있는 부모 테이블(`member_n`)의 키 값은 변경할 수 없다'는 규칙이다.

실제로 데이터를 넣고 이메일 변경을 시도한다고 가정해보자.

테스트 데이터 준비

```
-- 기존 이메일로 회원 가입
INSERT INTO member_n (email, password, name)
VALUES ('member1@old.com', 'hashed_password', '네이트');

-- 해당 회원이 상품 주문
INSERT INTO orders_n (member_email, ordered_at)
VALUES ('member1@old.com', NOW());
```

이메일 변경 시도 및 결과

이제 이 회원의 이메일 변경을 시도하면, 데이터베이스는 다음과 같은 오류를 내뱉으며 쿼리 실행을 거부할 것이다.

```
UPDATE member_n SET email = 'member1@new.com' WHERE email = 'member1@old.com';
```

[실행 결과]

```
Error Code: 1451. Cannot delete or update a parent row: a foreign key
constraint fails (`my_shop3`.`orders_n`, CONSTRAINT `fk_orders_member_n`
FOREIGN KEY (`member_email`) REFERENCES `member_n` (`email`))
```

이 오류 메시지는 우리에게 중요한 사실을 알려준다. `orders_n` 테이블이 `fk_orders_member_n`라는 외래 키 제약조건을 통해 `member_n` 테이블의 `member1@old.com` 값을 참조하고 있기 때문에, `member_n` 테이블의 PK를 함부로 바꿀 수 없다는 뜻이다.

이것이 바로 데이터베이스가 데이터의 **정합성(Consistency)**과 **무결성(Integrity)**을 지키는 방식이다. 만약 이 `UPDATE`가 성공했다면, 주문 데이터는 주인을 잃고 떠다니는 고아 데이터가 되었을 것이다. 데이터베이스의 기본 설정이 우리 시스템을 큰 재앙으로부터 보호해 준 것이다.

문제 2: 연쇄 업데이트와 시스템 부하

물론 "FK 제약조건을 잠시 끄고, 연관된 모든 테이블의 데이터를 다 찾아서 변경하면 될 것 같은데?" 라고 생각할 수 있다. 하지만 만약 이 회원이 수천 건의 주문, 수백 개의 게시글, 수십 개의 리뷰를 작성했다면? 단 한 번의 이메일 변경으로 인해 데이터베이스 전체에 걸쳐 수천 개의 레코드를 수정하는 끔찍한 연쇄 업데이트가 발생한다. 이는 데이터베이스에 엄청난 부하를 준다.

문제 3: 데이터의 역사성 훼손
데이터베이스가 마법처럼 모든 참조를 완벽하게 업데이트했다고 가정해 보자. `member_n` 테이블의 PK는 `member1@new.com`이 되었고, `orders_n` 테이블의 `member_email`도 모두 `member1@new.com`으로 바뀌었다. 이제 모든 문제가 해결된 것일까?

아니다. 더 교묘하고 심각한 문제가 발생한다. 바로 **데이터의 역사성**이 훼손된 것이다.

분명히 `member1` 회원은 과거 `member1@old.com` 이메일을 사용하던 시점에 주문을 했다. 그런데 이제 데이터베이스에는 그 주문을 `member1@new.com`이 한 것처럼 기록되어 있다. 이것은 명백한 **사실 왜곡**이다.

이게 왜 문제일까? 몇 달 뒤, 고객 지원팀에서 다급한 연락이 온다고 상상해 보라.
"고객님이 `member1@old.com` 시절에 주문했던 내역에 대해 문의하시는데, 우리 시스템에서는 `member1@old.com`으로 조회되는 주문이 하나도 없어요! 어떻게 된 거죠?"

데이터베이스의 모든 기록이 현재 시점의 이메일로 덮어씌워졌기 때문에, 과거의 특정 시점에 어떤 데이터가 유효했는지 추적할 방법이 사라진 것이다. 이는 장애 추적, 데이터 분석, 나아가 법적 분쟁 대응 등 신뢰성 있는 데이터가 필수적인 모든 영역에서 치명적인 문제가 된다.

문제 4: 외부 시스템과의 연동 문제
현대의 서비스는 혼자 동작하지 않는다. 우리 쇼핑몰도 주문 데이터를 외부 배송업체 시스템, 이메일 마케팅 솔루션, 데이터 분석 플랫폼 등 수많은 외부 시스템과 연동하고 있을 것이다.

우리가 `member1@old.com` 회원의 주문 정보를 외부 배송 시스템에 전달했다고 가정하자. 배송 시스템은 `member1@old.com`을 식별자로 사용해 배송 상태를 추적하고 있을 것이다.

그런데 우리 쇼핑몰 데이터베이스에서 이메일을 `member1@new.com`으로 변경해 버리면 어떻게 될까? 우리 시스템과 외부 배송 시스템 간의 **데이터 동기화가 깨진다**. 이제 두 시스템은 동일한 회원을 서로 다른 식별자(`member1@new.com` vs `member1@old.com`)로 인식하게 된다.

이 상태에서 고객이 배송 상태를 조회하거나 반품을 신청하면, 우리 시스템은 `member1@new.com`으로 배송 시스템에 정보를 요청하지만, 배송 시스템에는 해당 데이터가 존재하지 않아 오류가 발생할 것이다. 이 문제를 해결하려면 외부

시스템과 연동하는 모든 지점을 찾아 복잡한 데이터 동기화 로직을 추가로 개발해야만 한다. 비즈니스 로직(이메일)을 PK로 사용한 대가로, 시스템 전체의 복잡도가 기하급수적으로 증가하는 것이다. (이런 데이터 동기화를 맞추는 작업은 개발자에게는 정말 까다롭고 번거로운 일이다! 상상만 해도 끔찍하다.)

이처럼 자연 키는 '변경'이라는 시한폭탄을 품고 있다. 데이터베이스 내부의 정합성을 깨뜨릴 뿐만 아니라, 데이터의 역사성을 왜곡하고, 외부 시스템과의 연동까지 마비시킬 수 있는 매우 위험한 설계 방식이다.

'절대 변하지 않을 것'이라는 착각

"이메일 말고, 절대 변하지 않을 것 같은 로그인ID나 주민등록번호 같은 것을 사용하면 되지 않나요?"

- 예를 들어 회원 가입시에 `hello123` 같은 로그인 ID를 입력 받는다고 가정해보자. 이런 로그인 ID는 변하지 않을 것 처럼 보인다. 이 값이 정말 변하지 않을까? 예를 들어서 로그인 ID에 욕설이 포함되어 있다고 가정해보자. 그리고 다른 사용자들도 해당 로그인 ID를 볼 수 있다면? 이후에 이런 부분을 발견하면 어떻게 해야할까?
- 우리가 잘 아는 주민등록번호는 과연 변하지 않을까? 아니다. 실제로 주민등록번호도 나라에 신청해서 변경할 수 있다.

그렇다면 이 문제를 어떻게 해결해야 할까? 바로 다음에 배울 **대리 키(Surrogate Key)**가 그 정답이다.

자연 키 vs 대리 키2 - 대리 키

자연 키가 가진 '변경 가능성'이라는 치명적인 약점을 해결하기 위해 등장한 개념이 바로 **대리 키(Surrogate Key)**, 다른 말로는 **인조키(Artificial Key)**다.

대리 키의 핵심 아이디어는 간단하다. "비즈니스 로직과 완전히 무관한, 오직 데이터를 식별하기 위한 용도로만 존재하는, 임의의 값을 기본 키로 사용하는 것이다."

대리 키란 무엇인가?

대리 키의 존재 이유는 단 하나, **절대 변하지 않는다**는 것이다.
대리 키는 비즈니스와 아무런 관련이 없는, 시스템이 자동으로 생성해 주는 값이다. 보통 1, 2, 3, ... 과 같이 순차적으로 증가하는 정수나, `f47ac10b-58cc-4372-a567-0e02b2c3d479`와 같이 전역적으로 고유함이 보장되는 UUID(Universally Unique Identifier)를 사용한다.

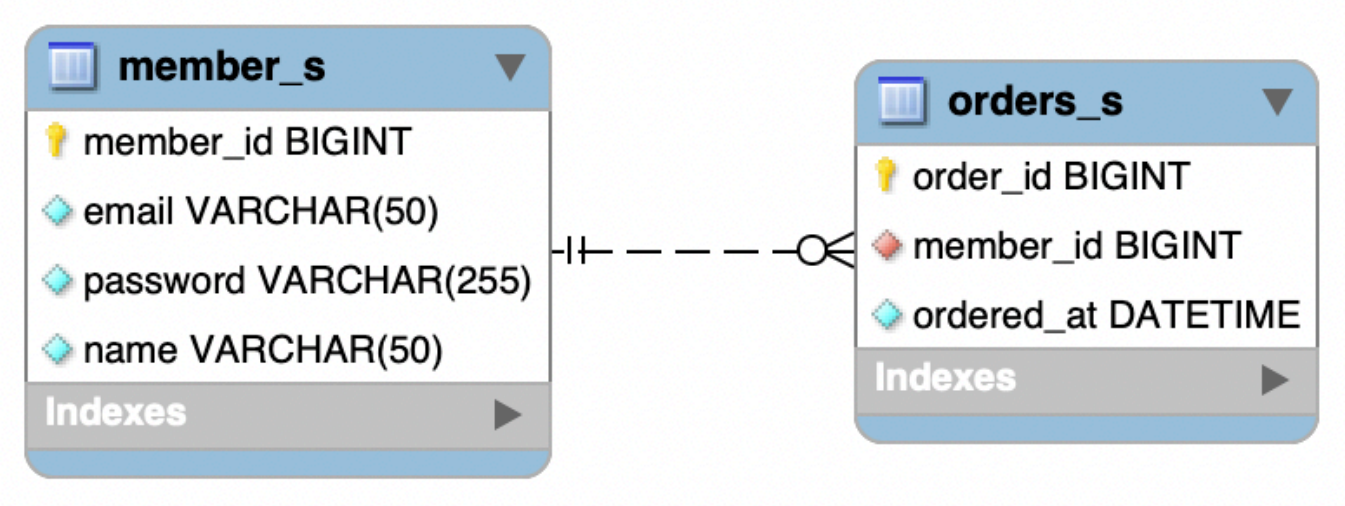
> **UUID - Universally Unique Identifier**
> 이름 그대로 전 세계에서 유일무이할 것으로 기대되는 값을 만들어내는 표준이다. 쉽게 이야기해서 아주 아주 큰 랜덤값이어서, 확률적으로 충돌이 거의 발생하지 않는다.
> 약 1초에 10억 개의 UUID를 100년 동안 계속 만들다 보면 한 번 정도 충돌할 수 있다.
> 따라서 혹시나 충돌하면 어떡하지?'라는 걱정은 사실상 하지 않아도 된다.

이제 올바른 설계로 이전의 이메일 변경 시나리오를 다시 해결해 보자.
앞서 만든 `member_n` 테이블 대신에 새로운 테이블 `member_s`를 만들자.
`member_id` 컬럼을 만들고, `AUTO_INCREMENT` 속성을 부여해서 대리 키를 사용해보자.

```
DROP TABLE IF EXISTS orders_s;
DROP TABLE IF EXISTS member_s;

-- 올바른 설계
CREATE TABLE member_s (
    member_id   BIGINT       NOT NULL AUTO_INCREMENT, -- PK (대리 키)
    email       VARCHAR(50) NOT NULL,                -- 비즈니스 로직
    password    VARCHAR(255) NOT NULL,
    name        VARCHAR(50)  NOT NULL,
    PRIMARY KEY (member_id),
    UNIQUE KEY uq_email (email) -- 고유해야 하는 자연 키는 UNIQUE로!
);

-- member_s 테이블을 안정적으로 참조하는 orders_s 테이블
CREATE TABLE orders_s (
    order_id    BIGINT       NOT NULL AUTO_INCREMENT,
    member_id   BIGINT       NOT NULL,                -- FK
    ordered_at  DATETIME     NOT NULL,
    PRIMARY KEY (order_id),
    CONSTRAINT fk_orders_member_s FOREIGN KEY (member_id) REFERENCES member_s
(member_id)
);

-- 기존 이메일로 회원 가입
INSERT INTO member_s (email, password, name)
VALUES ('member1@old.com', 'hashed_password', '네이트');

-- 앞서 저장한 회원 ID 임시 저장
SET @last_member_id = LAST_INSERT_ID();
```

```
-- 해당 회원이 상품 주문
INSERT INTO orders_s (member_id, ordered_at)
VALUES (@last_member_id, NOW());
```

- 대리 키(Surrogate Key)를 사용하는 예시이므로 학습 목적으로 테이블을 쉽게 구분하기 위해 뒤에 `_s`를 붙였다.
- `member_s` 테이블에 `member_id`라는 대리 키를 만들었다.
- `orders_s` 테이블은 이제 `member_id` FK를 사용한다.



대리 키는 어떻게 자연 키의 문제를 해결하는가?

위의 설계가 어떻게 자연 키의 문제들을 해결하는지 하나씩 살펴보자.

1. 영원히 변하지 않는 키

`member_id`는 회원이 가입하는 순서에 따라 `1`, `2`, `3`... 과 같이 자동으로 부여되는 값이다. 이 값은 비즈니스 로직과 아무런 관련이 없다. 회원이 이메일을 바꾸든, 이름을 바꾸든, 이사를 가든, `member_id`는 **절대로 변하지 않는다.** 데이터베이스 설계의 가장 중요한 원칙 중 하나인 '기본 키는 불변(Immutable)해야 한다'를 완벽하게 만족한다.

2. 손쉬운 이메일 변경

이제 고객 지원팀에서 이메일 변경 기능을 요청하면 어떻게 될까? 우리는 아주 간단하고 안전하게 기능을 구현할 수 있다.

`member1` 회원의 `member_id`가 `1`이라고 가정하자. 이 회원의 이메일을 `member1@old.com`에서 `member1@new.com`으로 변경하는 쿼리는 다음과 같다.

```sql
UPDATE member_s
SET email = 'member1@new.com'
WHERE member_id = 1;
```

이 쿼리는 `member_s` 테이블의 `email` 컬럼만 수정할 뿐, PK인 `member_id`는 전혀 건드리지 않는다. 따라서 이 회원과 연결된 수천 개의 `orders_s`, `posts`, `reviews` 테이블의 어떤 데이터도 수정할 필요가 없다. **연쇄 업데이트가 발생하지 않는다.** 데이터베이스에 부하를 주지 않으며, 데이터 정합성이 깨질 위험도 전혀 없다.
비즈니스 로직의 변경이 데이터 구조에 아무런 영향을 주지 않는, 완벽하게 **느슨하게 결합된(Loosely Coupled)** 이상적인 구조가 완성된 것이다.

member_s 테이블 조회
이제 `member_s` 테이블을 조회해서 이메일이 잘 변경되었는지 확인해 보자.

```sql
SELECT * FROM member_s;
```

[실행 결과]

member_id	email	password	name
1	member1@new.com	hashed_password	네이트

orders_s 테이블 조회
`member_s` 테이블의 `email`이 변경되어도 `orders_s` 테이블은 아무런 영향을 받지 않았다는 것을 확인해 보자.
`orders_s` 테이블은 오직 `member_id`만 바라보고 있기 때문이다.

```sql
SELECT * FROM orders_s;
```

[실행 결과]

order_id	member_id	ordered_at
1	1	주문 시각

결과를 보면 `orders_s` 테이블의 `member_id`는 `1`로 그대로 유지되고 있으며, 우리는 이 값을 통해 여전히 '네이트' 회원의 주문이라는 사실을 명확하게 알 수 있다. 이처럼 대리 키를 사용하면 비즈니스 요구사항 변경에 매우 유연하고 안정적으로 대처할 수 있다.

3. 비즈니스 로직의 유연성 확보

"이메일은 절대 변경할 수 없다"는 비즈니스 정책이 "이제부터 변경 가능하다"로 바뀌어도, 데이터베이스 설계는 아무런 영향을 받지 않는다. **비즈니스 로직과 식별자를 분리**했기 때문이다.

그럼 여기서 이메일의 중복을 막고 싶다면 어떻게 할까? PK가 아닌 `UNIQUE` 제약조건을 `email` 컬럼에 걸어주면 된다. 이렇게 하면 PK의 역할(행의 유일한 식별)과 비즈니스 규칙(이메일 중복 방지)을 명확하게 분리하여 각각의 역할에 충실한, 유연하고 안정적인 설계를 만들 수 있다.

자연 키의 올바른 사용법: UNIQUE 제약조건

그렇다면 자연 키는 이제 쓸모없는 것일까? 아니다. PK를 대리 키에게 넘겨주고, 본연의 역할에 집중하면 된다. 즉, **비즈니스적으로 고유해야 하는 모든 컬럼에 `UNIQUE` 제약조건을 설정**하는 것이다.

위의 `member_s` 테이블 설계에서 `email`에 `UNIQUE` 키를 설정하면, 우리는 다음의 모든 이점을 얻을 수 있다.

1. **관계의 안정성**: 불변의 대리 키 `member_id`가 모든 외래 키 관계의 중심을 잡아준다.
2. **데이터 무결성**: `UNIQUE` 제약조건이 `email`의 중복을 데이터베이스 차원에서 막아준다.
3. **조회 성능**: `UNIQUE` 제약조건을 설정하면 해당 컬럼에 자동으로 인덱스가 생성되므로 `email`을 통한 조회 성능도 보장된다.

현대 데이터베이스 설계의 표준 패턴: 대리 키-PK, 자연 키-UNIQUE

현대적인 데이터베이스 설계는 대리 키-PK, 자연 키-UNIQUE 방식이 사실상 표준이다.

- 대리 키(Surrogate Key)를 기본 키로 사용
- 자연 키(Natural Key)에는 UNIQUE 제약조건 적용

이 패턴은 관계 안정성, 데이터 무결성, 조회 성능이라는 세 가지를 모두 만족시키는 사실상의 표준 방식이다.
이 방식의 핵심은 역할의 분리이다.

기본 키(Primary Key)는 '대리 키(Surrogate Key)'에게

- **역할**: 다른 테이블과의 관계를 맺는 '연결고리' 역할에만 집중한다.

- **구현**: `order_id`, `member_id`처럼 비즈니스와 직접적인 관련이 없는, 시스템이 자동 생성하는 불변의 값 (주로 숫자나 UUID)을 사용한다.
- **이점**: 이 키는 절대 변하지 않으므로, 이 키를 참조하는 모든 외래 키(FK) 관계가 매우 안정적으로 유지된다.

자연 키(Natural Key)는 'UNIQUE 제약조건'으로

- **역할**: 비즈니스 로직상 **'데이터의 고유성'**을 보장하는 역할을 담당한다.
- **구현**: `email`, `주민등록번호`, `사용자 아이디`처럼 실제 비즈니스 의미를 가지며 중복되어서는 안 되는 컬럼에 UNIQUE 제약조건을 설정한다.
- **이점**:
  - **데이터 무결성**: 데이터베이스 시스템 차원에서 해당 값의 중복 입력을 원천적으로 차단하여 데이터의 신뢰성을 보장한다.
  - **성능 보장**: UNIQUE 제약조건이 설정된 컬럼에는 자동으로 인덱스(Index)가 생성되어, 해당 컬럼을 조건으로 데이터를 조회할 때 빠른 속도를 보장한다.

정리

"관계는 불변의 **대리 키 - PK**로 안정적으로 맺고, 비즈니스 데이터의 고유성은 **자연 키 - UNIQUE**로 확실하게 보장한다."

이처럼 각 키의 역할을 명확히 분리하여 장점만 취하는 것이 현대적인 데이터베이스 설계의 핵심이다.

> **실무 이야기 - 비즈니스 환경은 언젠가 변한다**
>
> 나의 경험을 하나 이야기하겠다. 레거시 시스템을 유지보수 할 일이 있었는데, 분석해보니 회원 테이블에 주민등록번호가 기본 키로 잡혀 있었다. 회원과 관련된 수많은 테이블에서 조인을 위해 주민등록번호를 외래 키로 가지고 있었고 심지어 자식 테이블의 자식 테이블까지 주민등록번호가 내려가 있었다. 문제는 정부 정책이 변경되면서 법적으로 주민등록번호를 저장할 수 없게 되면서 발생했다. 결국 데이터베이스 테이블은 물론이고 수많은 애플리케이션 로직을 수정해야 했다.
>
> 만약 데이터베이스를 처음 설계할 때부터 자연 키인 주민등록번호 대신에 비즈니스와 관련 없는 대리 키를 사용했다면 수정할 부분이 많지는 않았을 것이다. **기본 키의 불변 조건을 현재는 물론이고 미래까지 충족하는 자연 키를 찾기는 쉽지 않다**. 대리 키는 비즈니스와 무관한 임의의 값이므로 요구사항이 변경되어도 기본 키가 변경되는 일은 드물다.
>
> 대리 키를 기본 키로 사용하되 주민등록번호나 이메일처럼 자연 키의 후보가 되는 컬럼들은 필요에 따라 유니크 인덱스를 설정해서 사용하자.

자연 키 vs 대리 키3 - 성능 트레이드 오프

앞서 설계 관점에서 자연 키 보다는 대리 키를 사용하는 것이 좋다고 했다. 이번에는 성능 관점에서 둘을 비교해보자.
이번에는 자연 키와 대리 키를 선택했을 때 발생하는 성능상의 장단점, 즉 트레이드 오프에 대해 알아보자.

자연 키 사용의 장점

장점 - 조회 단순성: 비즈니스 요구사항을 보니 자연 키(예: `email`)만으로 데이터를 조회하는 경우가 대부분이라면, 조인(JOIN) 없이 해당 테이블에서 바로 원하는 정보를 찾을 수 있다. 예를 들어 주문 정보에서 회원의 `email`를 보고 싶을 때, `orders` 테이블에 `email`이 직접 저장되어 있다면 `member` 테이블을 조인할 필요가 없다.

예제: 주문 정보에서 회원 이메일 조회하기
앞서 자연 키의 문제점을 알아볼 때 만들었던 `orders_n` 테이블과, 대리 키를 사용해 올바르게 설계한 `orders_s` 테이블을 비교하며 직접 확인해 보자.

1. 자연 키(`member_email`)를 사용하는 경우
자연 키를 FK로 사용하는 `orders_n` 테이블에는 `member_email` 컬럼이 있어, `member_n` 테이블과 조인하지 않아도 회원의 이메일을 바로 알 수 있다.

```
SELECT order_id, member_email, ordered_at FROM orders_n;
```

[실행 결과]

order_id	member_email	ordered_at
1	member1@old.com	(주문 시각)

2. 대리 키(`member_id`)를 사용하는 경우
반면, 대리 키를 FK로 사용하는 `orders_s` 테이블에는 `member_id`만 저장되어 있다. 따라서 회원의 이메일을 확인하려면 반드시 `member_s` 테이블을 `JOIN` 해야 한다.

```
SELECT
    o.order_id,
    m.email AS member_email,
    o.ordered_at
FROM orders_s o
```

```
JOIN member_s m ON o.member_id = m.member_id;
```

[실행 결과]

order_id	member_email	ordered_at
1	member1@new.com	(주문 시각)

참고로 이전 예제에서 회원의 이메일을 `member1@new.com`으로 변경했기 때문에, 현재 시점의 이메일이 조회되는 것을 확인할 수 있다.

자연 키를 사용하는 것이 JOIN을 피할 수 있어 쿼리가 더 단순하고 빨라 보일 수 있다.
여기에 더해서, 자연 키 그 자체로 데이터를 검색하는 경우를 생각해 보자.

예제: 특정 이메일로 주문 정보 검색하기
'member1' 이라는 이름으로 시작하는 이메일을 사용하는 회원의 모든 주문을 찾는 시나리오를 가정해 보자.

1. 자연 키(`member_email`)를 사용하는 경우
`orders_n` 테이블에는 `member_email`이 있으므로 `member_n` 테이블을 방문할 필요 없이 `orders_n` 테이블만으로 검색을 완료할 수 있다.

```
SELECT * FROM orders_n WHERE member_email LIKE 'member1%';
```

[실행 결과]

order_id	member_email	ordered_at
1	member1@old.com	(주문 시각)

쿼리가 매우 단순하고, `orders_n` 테이블 하나만 접근하므로 성능 면에서도 유리하다.

2. 대리 키(`member_id`)를 사용하는 경우
`orders_s` 테이블에는 이메일 정보가 없으므로, 검색 조건인 이메일을 사용하려면 반드시 `member_s` 테이블과 JOIN 해야 한다.

```
```

```sql
SELECT
    o.order_id,
    o.member_id,
    o.ordered_at
FROM orders_s o
JOIN member_s m ON o.member_id = m.member_id
WHERE m.email LIKE 'member1%';
```

[실행 결과]

order_id	member_id	ordered_at
1	1	(주문 시각)

이처럼 자연 키를 검색 조건으로 자주 사용한다면, 해당 자연 키를 FK로 직접 가지고 있는 테이블의 조회 성능이 JOIN을 생략할 수 있어 더 좋을 수 있다.

자연 키 사용의 단점

단점1 - 외래 키 크기: 이것이 자연 키의 가장 치명적인 단점 중 하나다. 자연 키가 `VARCHAR(50)`처럼 긴 문자열이라고 가정해 보자. 이 키를 참조하는 모든 자식 테이블들(`orders`, `qna`, `reviews` 등 수많은 테이블)의 외래 키 컬럼도 똑같이 `VARCHAR(50)`이 되어야 한다.

- **공간 낭비**: 일반적인 대리 키로 사용하는 `BIGINT`(8바이트)에 비해 훨씬 많은 디스크 공간을 차지한다.
- **성능 저하**: 테이블과 인덱스의 크기가 커지면 메모리에 한 번에 올릴 수 있는 데이터 양이 줄어든다. 또한 조인 연산 시 비교해야 할 데이터의 크기가 커져 메모리, CPU 사용량도 늘어난다. 이는 전반적인 데이터베이스 성능 저하로 이어진다.

단점2 - 인덱스 단편화와 쓰기 성능 저하: `email`, `login_id`와 같은 자연 키는 일반적으로 알파벳순이나 가입 순서와 무관하게 생성된다. 예를 들어 `apple`, `mango`, `banana` 순서로 회원이 가입했다고 상상해 보자. 데이터베이스는 이 값들을 정렬된 상태로 인덱스에 저장해야 하므로, `apple` 다음에 `mango`를 넣었다가, 다시 그 사이에 `banana`를 삽입해야 한다.

- 이 과정에서 인덱스 페이지를 나누는 **페이지 분할(Page Split)**이 빈번하게 발생한다. 페이지 분할은 새로운 페이지를 할당하고 기존 데이터를 옮기는 복잡하고 비용이 큰 작업이다.
- 결과적으로 인덱스 내부에 빈 공간이 많이 생기는 **단편화(Fragmentation)**가 심해지고, 이는 쓰기 성능을 저하

시킨다.

> **페이지 분할, 단편화에 대한 자세한 내용은 데이터베이스 - 고급편에서 다룬다.**

대리 키 사용과 장단점

장점1 - 뛰어난 쓰기 성능: `AUTO_INCREMENT`나 `SEQUENCE`를 사용하는 대리 키는 **항상 이전 값보다 큰 숫자**를 순서대로 만들어낸다. 새로운 데이터는 인덱스 구조의 가장 마지막에 차례대로 추가(append)되기만 하면 된다.

- 이는 페이지 분할을 거의 일으키지 않으므로 인덱스 단편화 문제가 발생하지 않는다.
- 결과적으로 매우 빠르고 효율적인 쓰기 성능을 보장한다.

> **MySQL과 클러스터링 인덱스**
> MySQL의 기본 스토리지 엔진인 InnoDB 스토리지 엔진은 PK에 클러스터링 인덱스라는 특별한 인덱스를 사용한다. 클러스터링 인덱스는 `AUTO_INCREMENT`처럼 순서대로 생성되는 대리 키 방식에 최적화된 저장 성능을 제공한다.
> 클러스터링 인덱스에 대한 자세한 내용은 '실전 데이터베이스 고급편'에서 다룬다.

장점2 - 외래 키 크기 최소화: `BIGINT` 타입의 대리 키는 8바이트 고정 크기를 가진다. 이는 `VARCHAR(50)` 같은 가변 길이 문자열보다 훨씬 작고 효율적이다.

- 모든 자식 테이블에서 이 작은 크기의 외래 키를 사용하므로, 테이블과 인덱스의 전체적인 크기를 줄여 디스크 공간을 절약한다.
- 메모리 효율을 높이고, 조인 시 비교 연산이 빨라져 조회 성능까지 향상시킨다.

단점1 - 추가 조인 발생: 앞서 보았듯이 대리 키를 사용하면 특정 비즈니스 데이터를 얻기 위해 추가적인 조인이 필요할 수 있다. 예를 들어, `orders` 테이블에는 `member_id`만 저장되어 있다. 만약 주문 목록을 보면서 각 주문을 한 회원의 `email`을 함께 표시해야 한다면, 반드시 `member` 테이블을 조인해야만 한다.

```
-- 주문 ID 100번에 대한 회원의 email을 확인하려면 JOIN이 필수다.
SELECT m.email
  FROM orders o
  JOIN member m ON o.member_id = m.member_id
WHERE o.order_id = 100;
```

자연 키를 사용했다면 이 조인은 필요 없었을 것이다. 하지만 대부분의 경우, 이 조인으로 인한 약간의 성능 저하보다는 대리 키가 제공하는 다른 수많은 이점(쓰기 성능, 데이터 크기, 유연성)이 훨씬 크다.

단점2 - 추가 인덱스 필요: 대리 키를 기본 키(PK)로 사용한다는 것은, 원래 자연 키로 사용하려던 컬럼(예: `email`)을 이제 일반 컬럼으로 관리한다는 의미다. 하지만 사용자는 여전히 `email`로 로그인을 하고, `email`로 회원 검색을 해야 한다. 따라서 이 컬럼의 조회 성능을 보장하고 중복을 방지하기 위해 별도의 **UNIQUE 인덱스**를 반드시 생성해야 한다.

```
-- member 테이블에는 2개의 인덱스가 필요하다.
-- 1. 기본 키(PK) 인덱스: member_id
-- 2. 고유(UNIQUE) 인덱스: email
CREATE TABLE member (
    member_id BIGINT NOT NULL AUTO_INCREMENT,
    email  VARCHAR(50) NOT NULL,
    ...
    PRIMARY KEY (member_id),
    UNIQUE KEY uq_email (email)
);
```

인덱스가 하나 더 늘어난다는 것은 그만큼의 저장 공간을 추가로 사용하고, 데이터를 삽입/수정/삭제할 때마다 관리해야 할 인덱스가 하나 더 늘어난다는 의미다. 즉, 약간의 쓰기 성능 저하를 감수해야 한다.

정리하면 자연 키, 대리 키는 성능 관점에서 각각 장단점이 있지만, 대리 키를 기본 키(PK)로 사용하면 **데이터 모델의 안정성, 유연성, 그리고 쓰기 성능**이라는 더 큰 이점을 얻을 수 있기 때문에 실무에서는 대부분 대리 키를 기본 키로 선택한다.

자연 키 vs 대리 키4 - 현대적인 설계

현대적인 데이터베이스 설계에서는 자연 키와 대리 키 중에 어떤 것을 기본 키로 선택해야 할까?
결론부터 말하면, 거의 모든 경우에 대리 키 사용을 강력하게 권장한다.

자연 키는 비즈니스 로직과 데이터의 '신원(ID)'을 강하게 결합시킨다. 이는 앞서 본 예시처럼 비즈니스 로직이 변경될

때 데이터베이스의 근간을 흔들 수 있는 심각한 위험을 내포한다.

반면, **대리 키는 데이터의 신원(ID)을 비즈니스 로직으로부터 완전히 분리**시킨다. 덕분에 비즈니스 요구사항이 아무리 자유롭게 바뀌어도 데이터 모델의 안정성은 흔들리지 않는다. 회원의 이메일 변경? `member` 테이블의 `email` 컬럼 값만 바꾸면 그만이다. `member_id`는 그대로이므로, 이 회원을 참조하는 `orders`, `qna`, `review` 테이블은 아무런 영향을 받지 않는다. 이것이 바로 **느슨한 결합(Loose Coupling)**의 힘이다.

이러한 유연성과 안정성 때문에 현대적인 애플리케이션 설계 환경에서는 대리 키를 기본 전략으로 선택한다.

과거에는 자연 키를 사용하던 설계가 많았는데, 현대적인 데이터베이스 설계는 왜 이렇게 대리 키를 압도적으로 선호하게 되었을까? 그 배경을 한번 깊이 있게 파고들어 보자. 여기에는 기술의 발전과 애플리케이션 개발 방식의 근본적인 변화가 자리 잡고 있다.

과거: 데이터 중심의 시대와 자연 키

과거의 데이터베이스 설계는 애플리케이션보다 **데이터 그 자체의 독립성**에 더 큰 중점을 두었다. 데이터베이스가 시스템의 중심이었고, 여러 다른 애플리케이션을 이 중앙 데이터베이스에 직접 연결하여 데이터를 사용하는 경우가 많았다.

이런 환경에서는 데이터 자체만으로 의미를 가지는 **자연 키**가 매우 합리적인 선택이었다.

- **직관성**: `주민등록번호`, `학번`, `이메일`과 같은 자연 키는 그 값만 봐도 어떤 데이터인지 바로 알 수 있었다. DB 관리자(DBA)가 SQL 클라이언트로 직접 데이터를 조회하고 관리하는 일이 많았기 때문에 이런 직관성은 큰 장점이었다.
- **저장 공간 효율성**: 지금이야 저장 공간 비용이 매우 저렴해졌지만, 과거에는 디스크 비용이 비쌌다. 자연 키를 사용하면 별도의 ID 컬럼을 추가할 필요가 없으므로, 아주 약간이라도 저장 공간을 아낄 수 있었다.
- **'데이터 모델링 순수주의'**: 데이터 모델링 관점에서, 엔티티(Entity)는 고유하게 식별 가능한 속성(Attribute)을 가져야 한다는 원칙이 있었다. 이 원칙에 따라 비즈니스적으로 의미 있는 식별자(자연 키)를 기본 키로 삼는 것이 '올바른' 설계라고 여겨졌다.

과거 데이터 중심 설계의 중요한 원칙은 "현실 세계의 데이터를 가장 논리적이고 자연스럽게 표현하는 것"이었다. 이를 위해 모델러들은 각 테이블(엔티티)의 **자연스러운 식별자, 자연 키(Natural Key)**를 찾는 데 많은 노력을 기울였다. 그리고 과거에는 비즈니스 로직의 변경 속도가 지금처럼 빠르지 않았다. 한번 정해진 `주민등록번호`나 `학번` 같은 식별자는 거의 바뀌지 않을 것이라는 믿음이 있었다. 따라서 자연 키의 '불변성'에 대한 가정이 어느 정도는 통용될 수 있었다.

> **레거시 프로젝트**
> 한번 설계된 데이터베이스는 아주 오랜기간 유지된다.

> 특히 오래된 레거시 프로젝트들은 자연 키를 사용하는 경우가 많다.

현대: 애플리케이션 중심의 시대와 대리 키

하지만 인터넷이 보편화되고 애플리케이션의 복잡성이 기하급수적으로 증가하면서 상황은 완전히 달라졌다. 이제 시스템의 중심은 데이터베이스가 아닌 **애플리케이션**으로 이동했다.

1. 비즈니스 요구사항의 폭발적인 변화 속도

스타트업과 애자일(Agile) 개발 방식이 대세가 되면서, '비즈니스 로직은 언제든지 바뀔 수 있다'는 것이 기본 전제가 되었다. 어제까지 회원을 식별하던 유일한 값이 이메일이었는데, 내일부터는 소셜 로그인을 도입하면서 이메일이 없는 회원이 생길 수도 있다.

이런 환경에서 자연 키는 시한폭탄과 같다. 이메일을 PK로 사용했다면, 소셜 로그인 기능을 도입하는 순간 데이터베이스 설계의 근간부터 다시 고민해야 하는 대참사가 발생한다. 하지만 비즈니스와 아무 관련 없는 숫자 ID, 즉 **대리 키**를 사용했다면 어떨까? 이메일 정책이 어떻게 바뀌든, `member_id`는 영원히 그 자리에 그대로 있다. 애플리케이션의 비즈니스 로직만 수정하면 될 뿐, 데이터 모델은 전혀 영향을 받지 않는다.

이처럼 현대의 데이터베이스 설계는 **변경에 유연하게 대처하는 능력(Flexibility)**이 매우 중요하다.

2. ORM 기술의 등장과 패러다임 전환

JPA(Java Persistence API), Hibernate, Entity Framework 같은 **ORM(Object-Relational Mapping)** 기술의 등장도 대리 키 사용을 가속화했다.

ORM은 '애플리케이션의 객체'와 '관계형 데이터베이스의 테이블' 사이의 불일치를 해소하고 자동으로 연결해주는 기술이다. 개발자가 SQL을 직접 작성하지 않아도 객체를 통해 데이터베이스 작업을 할 수 있도록 돕는 '똑똑한 통역사'라고 생각할 수 있다.

개발자는 자바나 C# 같은 객체 지향 언어로 비즈니스 로직을 만든다. 하지만 객체 지향 언어로 개발을 하더라도 데이터는 결국 관계형 데이터베이스에 저장해야 한다. 따라서 관계형 데이터베이스의 테이블 구조에 맞춰 SQL을 작성해야한다. 이 두 가지 방식은 근본적으로 다르기 때문에, 개발자는 항상 '객체 세계'와 '데이터베이스 세계' 사이에서 객체를 SQL로 변경하고, 또 SQL의 결과를 객체로 변경하는 지루한 번역 작업을 계속해야만 했다.

ORM은 바로 이 귀찮고 반복적인 번역 작업을 대신 처리해주는 기술이다.

ORM 이전: 개발자가 모든 SQL을 직접 다루던 시대

ORM이라는 통역사가 없던 시절, 개발자는 모든 SQL을 직접 작성해야 했다.

애플리케이션에서 `member` 객체를 다루기 위해, 개발자는 `회원 저장소`와 같은 클래스를 만들고 그 안에 CRUD(생성, 조회, 수정, 삭제) 기능을 위한 SQL을 하나하나 직접 작성해야 했다.

예를 들어, 다음과 같은 `member` 객체가 있다고 가정해 보자.

```
class Member {
    private Long memberId;
    private String loginId;
    private String password;
    private String name;
    private String email;
    private DateTime created_at;
    // ...
}
```

이 객체를 DB에 저장하고 관리하는 코드는 다음과 같다.

회원 저장 (Create)

```
public Member save(Member member) {
    String sql = "INSERT INTO member(login_id, password, name, email,
address, created_at) VALUES (?, ?, ?, ?, ?, NOW())";
    // 코드를 통해 sql을 실행한다.
    return member;
}
```

회원 조회 (Read)

```
public Member findById(Long memberId) {
    String sql = "SELECT member_id, login_id, name, email, address,
created_at " +
                 "FROM member WHERE member_id = ?";
    // 코드를 통해 sql을 실행한다.
    return new Member(...);
}
```

회원 수정 (Update)

```
public void update(Member member) {
    String sql = "UPDATE member SET name = ?, email = ?, address = ? " +
                 "WHERE member_id = ?";
    // 코드를 통해 sql을 실행한다.
}
```

회원 삭제 (Delete)

```
public void delete(Member member) {
    String sql = "DELETE FROM member WHERE member_id = ?";
    // 코드를 통해 sql을 실행한다.
}
```

결과적으로 이 `member` 객체를 데이터베이스에 저장(INSERT)하거나, 특정 회원의 이름을 변경(UPDATE)하려면, 개발자는 다음과 같은 수 많은 SQL 구문을 직접 만들어서 애플리케이션 코드에 추가해야 했다.

- **회원 저장 (Create)**: `INSERT INTO member(login_id, ...) VALUES(?, ...)`
- **회원 조회 (Read)**: `SELECT member_id, ... FROM member WHERE member_id = ?`
- **회원 수정 (Update)**: `UPDATE member SET name = ?, ... WHERE member_id = ?`
- **회원 삭제 (Delete)**: `DELETE FROM member WHERE member_id = ?`

객체의 필드가 10개라면, `INSERT` 문에 10개의 컬럼과 10개의 물음표를, `SELECT` 문에 10개의 컬럼을 나열해야 한다.
만약 객체에 필드 하나가 추가되거나 변경되면 이 모든 SQL을 다시 찾아서 수정해야 하는 끔찍한 일이 발생한다.

ORM의 등장: 똑똑한 통역사가 알아서 처리하는 시대
JPA와 같은 ORM 기술을 사용하면, 개발자는 더 이상 단순한 CRUD를 위해 SQL을 직접 작성하지 않는다. 그저 객체를 ORM에게 넘겨주기만 하면, ORM이 해당 객체에 맞는 최적의 SQL을 대신 생성해서 데이터베이스와 통신한다.

예를 들면 그저 통역사에게 "이 `member` 객체를 DB에 저장해 줘" 라고 말하기만 하면 된다. 그러면 ORM 통역사가 알아서 `INSERT` SQL을 생성하고 실행한다.

```

```

```java
// JPA 통역사를 사용하는 가상 코드
// 회원 저장
public void save(Member member) {
    // "이 객체 저장해 줘" 라고 말하면 끝. INSERT SQL을 자동 생성해서 실행
    jpa.persist(member);
}

// 회원 삭제
public void delete(Member member) {
    jpa.remove(member); // 이 한 줄이면 DELETE SQL이 자동으로 생성 및 실행된다.
}

// 회원 조회
public member findById(Long memberId) {
    // "member 타입으로, ID가 memberId인 객체 찾아 줘" 라고 말하면 끝.
    // SELECT SQL을 자동 생성해서 실행
    return jpa.find(Member.class, memberId);
}

// 회원 이름 수정
public void updateName(Long memberId, String newName) {
    Member member = jpa.find(Member.class, memberId); // 먼저 객체를 찾아서
    member.setName(newName); // 객체의 이름만 바꾸면 끝!, UPDATE SQL을 자동 생성해서 실행
}
```

코드를 보면 이전에 작성한 SQL이 모두 사라진 것을 알 수 있다. 그리고 여기서는 JPA라는 ORM 기술이 이런 SQL을 대신 작성하고 실행한다. 따라서 이전과 비교해 개발 생산성이 비약적으로 증가한다.

여기서 `UPDATE` 코드를 눈여겨보자. `update` 라는 명령조차 없다. 개발자는 그저 회원을 찾아서 객체의 이름만 바꿨을 뿐이다. 어떻게 이게 가능할까?

JPA라는 똑똑한 통역사는 자신이 찾아준 객체를 예의주시하고 있다. 그러다 개발자가 객체의 값을 바꾸면, "어? 주인이 객체 상태를 바꿨네? 그럼 나중에 데이터베이스에도 반영해줘야겠다" 라고 기억해둔다. 그리고 모든 작업이 끝나는 시점에 **자동으로** `UPDATE` **SQL을 만들어서 실행**해준다.

ORM과 대리 키

그렇다면 "이 똑똑한 통역사(ORM)는 도대체 무슨 수로 회원 정보를 기억하고, 내용이 바뀐 것을 알아채는 걸까?"

정답은 **PK(기본 키)**에 있다. ORM은 자기가 관리해야 할 회원들의 **'관리 명단'**을 가지고 있다. 그리고 이 명단에서 각 회원을 구분하는 유일한 값으로 바로 **PK**를 사용한다.

이 똑똑한 통역사(ORM)가 원활하게 일하려면 한 가지 중요한 전제 조건이 필요하다. 바로 **'절대 변하지 않는 고유 식별자'**이다.

여기 도서관이 하나 있다고 상상해보자.

- **도서관**: 데이터베이스
- **사서**: ORM (JPA)
- **책**: 객체 (`BOOK` 객체)
- **도서관리번호**: **대리 키** (`book_id`)
- **책 제목**: **자연 키** (`book_name`)

사서(ORM)는 모든 책(객체)에 고유하고 절대 변하지 않는 '도서관리번호'(`book_id`)를 붙여서 관리한다. 덕분에 책의 제목(`book_name`)이 바뀌거나, 책에 낙서를 해도 사서는 도서관리번호만으로 어떤 책인지 정확하게 알 수 있다.
"아, 3번 책의 제목이 'A'에서 'B'로 바뀌었구나" 라고 쉽게 인지하고 수정(`UPDATE`)하면 그만이다.

그런데 만약 '책 제목'(`book_name`)을 도서관리번호처럼 사용한다면 어떻게 될까? 즉, `book_name`인 자연 키를 PK로 사용한 경우다.
어떤 사람이 'JPA 프로그래밍' 이란 책을 빌려갔는데, 책의 제목을 '스프링 프로그래밍'으로 수정하고 반납했다.
이때 사서(ORM)는 대혼란에 빠진다. '스프링 프로그래밍'이라는 책은 우리 도서관에 없던 책인데?
결국 책의 신원(ID) 자체가 바뀌어버렸기 때문에 사서는 이 책을 추적할 수 없게 된다.

이것이 바로 ORM이 대리 키를 압도적으로 선호하는 이유다.

대리 키는 비즈니스와 무관하게 절대 변하지 않는 '관리 번호' 역할을 하므로 ORM 통역사가 객체를 안정적으로 추적하고 관리할 수 있게 해준다. 반면, 언제든 바뀔 수 있는 비즈니스 속성인 자연 키는 ORM을 혼란에 빠뜨릴 수 있다.

이러한 이유로 JPA와 같은 ORM 기술은 **변경 불가능한(Immutable)** 대리 키 사용을 강력하게 권장한다.

현대 애플리케이션 개발은 ORM을 통해 생산성을 극대화한다. 그리고 ORM의 성능과 안정성을 100% 활용하기 위한 최고의 파트너가 바로 **대리 키**다. 이 둘의 조합은 개발자가 SQL이라는 번거로운 번역 작업에서 벗어나 비즈니스 로직 구현에만 집중할 수 있도록 해주는 핵심 기능이다.

3. 분산 시스템과 마이크로서비스 아키텍처(MSA)

최근에는 하나의 거대한 애플리케이션을 여러 개의 작은 서비스로 나누는 마이크로서비스 아키텍처가 널리 쓰이고 있다. '회원 서비스', '주문 서비스', '상품 서비스'가 각각 독립된 데이터베이스를 가질 수 있다.

이런 환경에서 `email` 같은 자연 키를 여러 서비스에 걸쳐 일관되게 사용하고 관리하는 것은 거의 재앙에 가깝다. '회

원 서비스'에서 이메일 정책이 바뀌면, '주문 서비스'를 비롯한 다른 모든 서비스도 그 변경에 영향을 받기 때문이다.

하지만 각 서비스가 외부에는 `member_id`라는 절대 변하지 않는 안정적인 대리 키 값으로만 소통한다면 어떨까? '회원 서비스' 내부에서 이메일이든, 전화번호든, 어떤 것으로 회원을 식별하든 다른 서비스는 전혀 신경 쓸 필요가 없다. 오직 `member_id` 값만 알면 된다. 이는 서비스 간의 **결합도(Coupling)**를 낮추어 시스템 전체의 안정성과 확장성을 크게 높여준다.

결론: 현대적 설계의 핵심은 '느슨한 결합'
정리하자면, 과거의 설계가 **데이터의 의미와 직관성**을 중시했다면, 현대의 설계는 **비즈니스 변화에 대한 유연성과 시스템의 안정성**을 최우선으로 고려한다.

대리 키는 비즈니스 로직이라는 변화무쌍한 세계로부터 데이터의 '신원(ID)'이라는 가장 중요한 본질을 지켜주는 **완벽한 안전장치** 역할을 한다. 이 안전장치 덕분에 우리는 비즈니스 로직은 비즈니스 로직대로, 데이터는 데이터대로 각자의 역할에 충실하며 안정적으로 성장하는 시스템을 만들 수 있다. 이것이 바로 현대적인 설계가 '느슨한 결합(Loose Coupling)'을 추구하며 대리 키를 압도적으로 선호하는 이유다. 실무에서는 고민의 여지 없이 대리 키를 사용하는 것이 정답에 가깝다.

> **대리 키-PK, 자연 키-UNIQUE**
> 현대적인 데이터베이스 설계는 대리 키-PK, 자연 키-UNIQUE 방식이 사실상 표준이다.
> 대리 키-PK, 자연 키-UNIQUE 이것 딱 하나만 기억하자.

복합키 설계

지금까지 우리는 하나의 컬럼으로 행을 식별하는 기본 키(단일키)에 대해 집중적으로 알아보았다. 하지만 경우에 따라서는 **하나의 컬럼만으로는 행을 고유하게 식별할 수 없는 상황**이 발생한다. 이때 등장하는 것이 바로 **복합키(Composite Key)**다.

복합키는 왜 필요한가? - 식별자의 조합

복합키가 왜 필요한지 이해하려면, 먼저 데이터 모델링 과정에서 '현실 세계의 데이터를 어떻게 논리적으로 표현할 것인가'라는 근본적인 질문으로 돌아가야 한다. 모델러들은 각 테이블(엔티티)의 **자연스러운 식별자(Natural Key)** 즉 **자연키**를 찾기 위해 노력하는데, 이 식별자가 항상 단일 컬럼으로 표현되지는 않기 때문이다.

예를 들어 '영화 예매' 시스템을 생각해보자. 우리는 먼저 '영화'와 '상영 정보', 그리고 '예매'라는 개념을 명확히 구분해야 한다.
예제를 단순화하기 위해 상영관은 서로 다른 영화를 보여준다고 가정하자.

1. **상영 정보(Screening)**: 특정 영화가 특정 날짜, 특정 시간에 상영되는 것을 의미한다. 예를 들어 '매트릭스1'이라는 **영화 제목**만으로는 특정 상영을 식별할 수 없다. 왜냐하면 '매트릭스1'은 여러 날짜와 시간에 걸쳐 수없이 많이 상영되기 때문이다. 이 상영을 유일하게 특정하려면 **'영화 제목'과 '상영 시작 시간'**이 함께 필요하다. 즉, '2025년 8월 31일 19시 30분에 시작하는 매트릭스1'이라고 해야 비로소 하나의 **'상영 정보'**가 명확히 식별된다.

2. **좌석 예매(Reservation)**: 이제 이 '상영 정보'에 좌석을 예매하는 상황을 생각해보자. '2025년 8월 31일 19시 30분 매트릭스1'라는 정보만으로는 예매를 특정할 수 없다. 왜냐하면 해당 상영 정보에는 수많은 좌석(F8, F9, G8 등)이 존재하고, 각 좌석마다 다른 관객이 예매하기 때문이다. 따라서 하나의 예매를 고유하게 식별하려면, **어떤 상영의 어떤 좌석**인지를 명시해야 한다.

결론적으로 '좌석 예매'라는 데이터를 식별하기 위한 가장 자연스러운 식별자는 `{영화 제목, 상영 시작 시간, 좌석 번호}`라는 세 가지 속성의 조합이 된다. 이처럼 엔티티의 본질 자체가 복합적인 식별자를 요구하는 상황에 복합키가 필요하다. 복합키는 두 개 이상의 컬럼을 함께 묶어 유일성을 확보하는 방식으로, 전통적 모델링에서 자연 키를 표현하는 핵심적인 방법이었다.

복합키를 사용한 테이블 설계

위 개념을 바탕으로, 복합키를 사용한 '영화 예매' 테이블을 설계해 보자. 여기서 '예매' 한 건을 식별하기 위한 기본 키는 `{영화 제목, 상영 시작 시간, 좌석 번호}`의 조합이 될 것이다.

테이블 구조 예시 (SQL)

```sql
DROP TABLE IF EXISTS popcorn_order_c;
DROP TABLE IF EXISTS movie_reservation_c;

CREATE TABLE movie_reservation_c (
    movie_title      VARCHAR(100) NOT NULL, -- 영화 제목
    screening_dt     DATETIME     NOT NULL, -- 상영 시작 시간
    seat_number      VARCHAR(10)  NOT NULL, -- 좌석 번호
    reserver_name    VARCHAR(50)  NOT NULL, -- 예매자 이름

    -- 복합키: movie_title + screening_dt + seat_number를 기본 키로 설정
    PRIMARY KEY (movie_title, screening_dt, seat_number)
```

```
);
```

- 복합키(Composite Key)를 사용하는 예시이므로 학습 목적으로 테이블을 쉽게 구분하기 위해 뒤에 `_c`를 붙였다.

이 테이블의 기본 키는 `movie_title`, `screening_dt(datetime 줄임)`, `seat_number` 세 컬럼의 조합이다.

- `{movie_title, screening_dt}`: 이 조합은 하나의 **상영 정보**를 식별한다.
- `{movie_title, screening_dt, seat_number}`: 이 조합은 해당 상영의 특정 좌석에 대한 **유일한 예매** 한 건을 식별한다.

예를 들어, '매트릭스1', '2025-08-31 19:30:00', 'F8'이라는 조합은 단 하나의 예매만을 가리킨다. 데이터베이스는 이 세 컬럼의 조합이 중복되어 삽입되는 것을 막아 데이터의 유일성을 보장한다. 이 키는 비즈니스 의미(어떤 영화 상영의 어떤 좌석)를 직접 반영하므로 자연 키의 성격을 띤다.

자연 키를 복합키로 사용하는 예시에서 발견되는 문제점

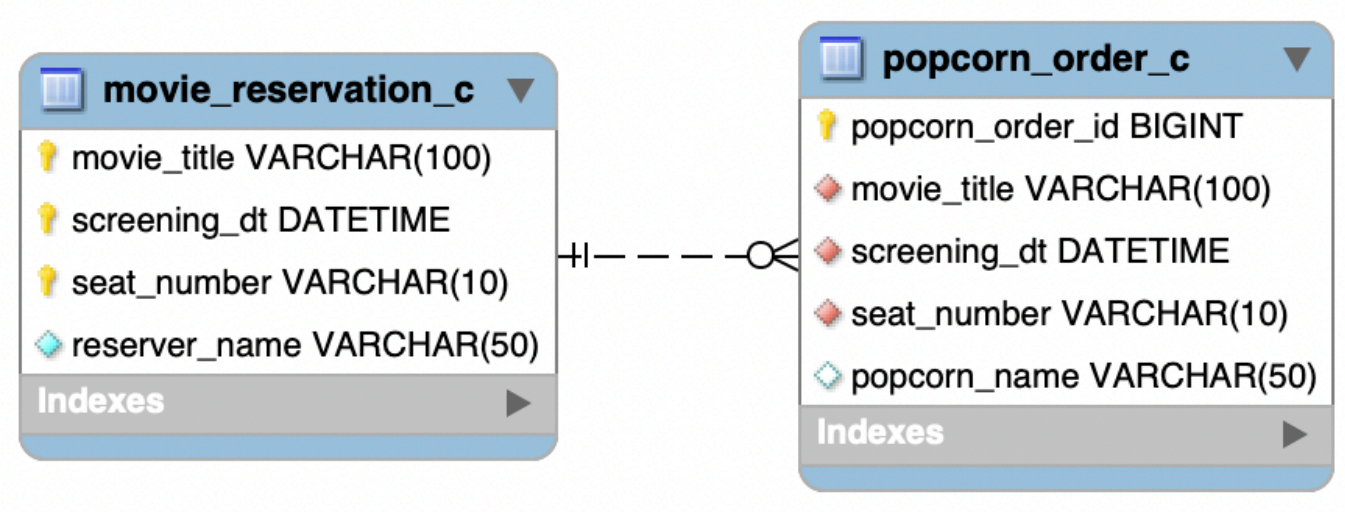
위 영화 예매 예시에서 복합키를 사용하면 여러 문제점이 드러난다.

- **변경 가능성으로 인한 불안정성**: 복합키가 자연 키 기반이므로 비즈니스 변화에 취약하다. 예를 들어, 영화관 사정으로 상영 시간(`screening_dt`)이 10분 지연되거나, 영화 제목(`movie_title`)에 오타가 있어 '매트릭사 1'에서 '매트릭스 1'로 수정해야 한다면 기본 키 자체가 바뀌어야 한다. 이 경우, 이 예매 정보를 참조하는 다른 테이블(예: '예매별 팝콘 주문' 테이블)의 외래 키도 모두 업데이트되어야 하며, 연쇄적인 데이터 수정이 발생한다. 이는 데이터 무결성을 위반할 위험이 크고, 연쇄 변경에 따른 시스템 부하도 증가시킨다.

- **외래 키 참조의 복잡성과 크기 증가**: 다른 테이블에서 이 복합키를 외래 키로 참조할 때, 모든 컬럼(`movie_title`, `screening_dt`, `seat_number`)을 복제해야 한다. 예를 들어, 특정 예매 좌석에 연결된 '팝콘 주문' 테이블에서 외래 키를 설정해보자.

```
CREATE TABLE popcorn_order_c (
    popcorn_order_id BIGINT       NOT NULL AUTO_INCREMENT,
    movie_title      VARCHAR(100) NOT NULL, -- FK1
    screening_dt     DATETIME     NOT NULL, -- FK2
    seat_number      VARCHAR(10)  NOT NULL, -- FK3
    popcorn_name     VARCHAR(50),
    PRIMARY KEY (popcorn_order_id),
    FOREIGN KEY (movie_title, screening_dt, seat_number)
```

```
        REFERENCES movie_reservation_c (movie_title, screening_dt,
seat_number)
);
```



이로 인해 `popcorn_order_c` 테이블의 크기가 불필요하게 커지고, 조인(JOIN) 쿼리가 복잡해진다. 여러 타입(`VARCHAR`, `DATETIME`)의 조합은 저장 공간을 더 많이 차지하며, 인덱스 효율성을 떨어뜨려 쓰기/조회 성능 저하의 원인이 될 수 있다.

- **조회와 관리의 어려움**: 복합키는 직관적이지 않다. 단일 키처럼 간단히 `WHERE id = 1`과 같이 조회할 수 없어, 쿼리가 길어지고 오류가 발생하기 쉽다. 또한, 실무에서 자주 사용하는 ORM(Object-Relational Mapping) 도구(예: JPA, Hibernate)에서 복합키를 다루려면 별도의 식별자 클래스를 만드는 등 추가적인 매핑 로직이 필요해 개발 복잡도가 증가한다.

- **확장성 문제**: 시스템이 성장하며 예매를 식별하는 데 '상영관 번호' 같은 새로운 조건이 추가된다면, 기본 키의 구성이 바뀌어야 한다. 이는 테이블 구조의 대대적인 변경을 의미하며, 관련된 모든 테이블과 쿼리에 영향을 미치는 큰 작업이 된다.

조인 쿼리 복잡성 예시

예를 들어, 특정 팝콘 주문(`popcorn_order_id`가 `101`인)을 한 사람의 이름(`reserver_name`)을 찾기 위해 `popcorn_order_c`와 `movie_reservation_c` 두 테이블을 조인하는 상황을 생각해보자. 먼저 샘플 데이터를 입력하고, 복합키를 사용한 조인 쿼리를 실행해 보자.

```
-- 샘플 데이터 삽입
INSERT INTO movie_reservation_c (movie_title, screening_dt, seat_number,
reserver_name)
```

```
VALUES ('매트릭스1', '2025-08-31 19:30:00', 'F8', '네이트');

INSERT INTO popcorn_order_c (popcorn_order_id, movie_title, screening_dt,
seat_number, popcorn_name)
VALUES (101, '매트릭스1', '2025-08-31 19:30:00', 'F8', '카라멜/치토스 팝콘');
```

```
SELECT
    mr.reserver_name,
    po.popcorn_order_id,
    po.popcorn_name
FROM
    popcorn_order_c po
JOIN
    movie_reservation_c mr
ON
    po.movie_title = mr.movie_title
    AND po.screening_dt = mr.screening_dt
    AND po.seat_number = mr.seat_number
WHERE
    po.popcorn_order_id = 101;
```

[실행 결과]

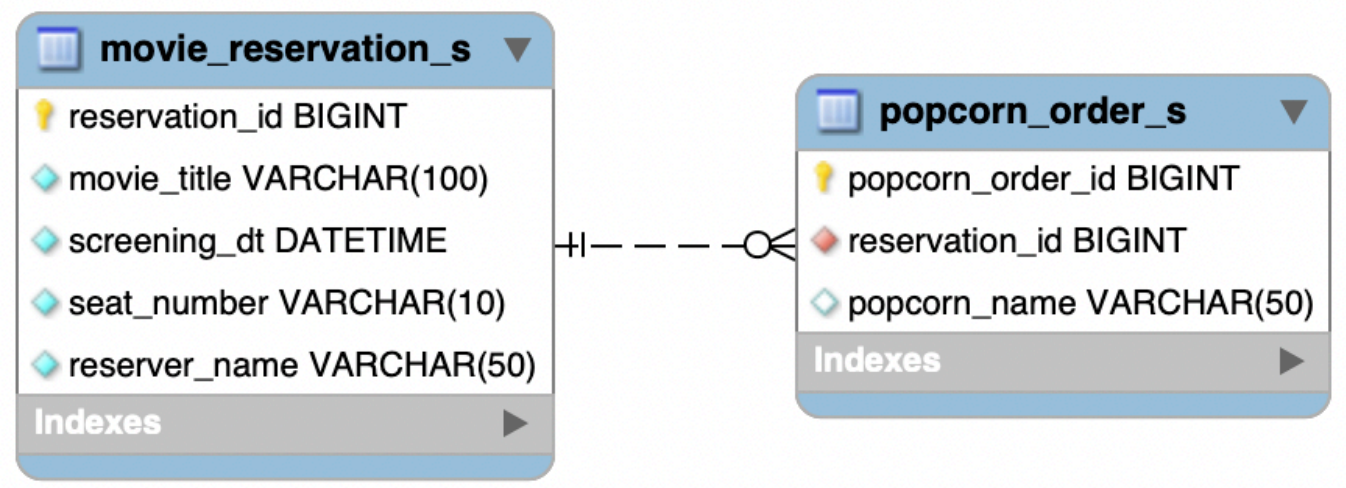
reserver_name	popcorn_order_id	popcorn_name
네이트	101	카라멜/치토스 팝콘

조인 조건절(`ON`)에서 `movie_title`, `screening_dt`, `seat_number` **세 개의 컬럼을 모두 비교**해야 한다. 지금은 컬럼이 세 개뿐이지만, 만약 네 개 이상의 컬럼으로 구성된 복합키라면 조인 조건은 훨씬 더 길고 복잡해질 것이다.

대안 - 대리 키

복합키의 문제점을 해결하는 주요 대안은 앞서 살펴본 대리 키(Surrogate Key)를 기본 키로 사용하는 것이다. 대리 키는 비즈니스와 무관한 자동 생성 값(예: `AUTO_INCREMENT`를 사용한 `BIGINT` 타입의 ID)을 활용하며, 불변성을 보장한다. 그리고 대리 키는 간단하게 하나의 컬럼만 사용한다.

위 영화 예매 예시를 대리 키로 재설계해보자.



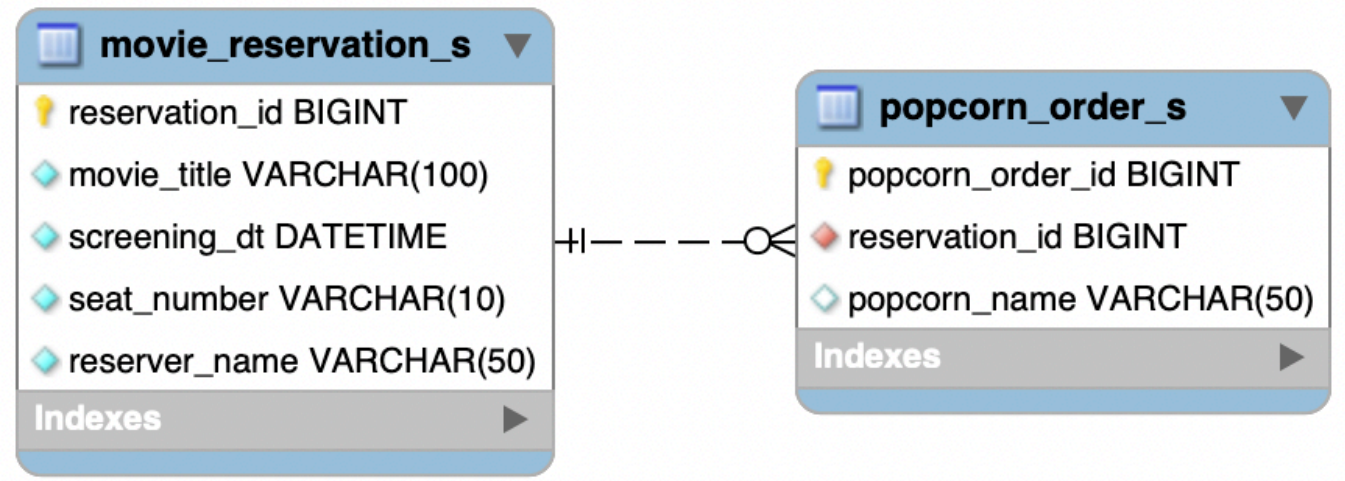
테이블 구조 예시 (SQL)

여기서는 대리 키를 사용하는 새로운 테이블(`_s` 접미사 사용)을 생성한다.

```
DROP TABLE IF EXISTS popcorn_order_s;
DROP TABLE IF EXISTS movie_reservation_s;

-- 예매 테이블 (대리 키 사용)
CREATE TABLE movie_reservation_s (
    reservation_id   BIGINT       NOT NULL AUTO_INCREMENT, -- 대리 키 PK
    movie_title      VARCHAR(100) NOT NULL,
    screening_dt     DATETIME     NOT NULL,
    seat_number      VARCHAR(10)  NOT NULL,
    reserver_name    VARCHAR(50)  NOT NULL,
    PRIMARY KEY (reservation_id),
    -- 자연 키 부분에 UNIQUE 제약으로 데이터 무결성 보장
    UNIQUE KEY uq_movie_reservation (movie_title, screening_dt, seat_number)
);

-- 팝콘 주문 테이블 (대리 키 참조)
CREATE TABLE popcorn_order_s (
    popcorn_order_id BIGINT      NOT NULL AUTO_INCREMENT,
    reservation_id   BIGINT      NOT NULL, -- 단순화된 FK
    popcorn_name     VARCHAR(50),
    PRIMARY KEY (popcorn_order_id),
    FOREIGN KEY (reservation_id) REFERENCES movie_reservation_s
(reservation_id)
);
```

- 대리 키(Surrogate Key)를 사용하는 예시이므로 학습 목적으로 테이블을 쉽게 구분하기 위해 뒤에 `_s`를 붙였다.

- `movie_reservation_s` 테이블은 `reservation_id`라는 대리 키를 기본 키로 가진다. 그리고 비즈니스 유일성을 보장해야 하는 `{movie_title, screening_dt, seat_number}` 조합에는 `UNIQUE` 제약조건을 걸어 데이터 무결성을 지킨다.
- `popcorn_order_s` 테이블은 이제 `reservation_id`라는 단 하나의 `BIGINT` 컬럼만 외래 키로 참조하면 된다.



단순화된 조인 쿼리 예시

이제 대리 키를 사용하여 동일한 조회(특정 팝콘 주문을 한 사람의 이름 찾기)를 수행해보자.

```
-- 샘플 데이터 삽입
INSERT INTO movie_reservation_s (movie_title, screening_dt, seat_number,
reserver_name)
VALUES ('매트릭스1', '2025-08-31 19:30:00', 'F8', '네이트');

-- movie_reservation_s 테이블에 방금 입력된 reservation_id는 1이다.
INSERT INTO popcorn_order_s (reservation_id, popcorn_name)
VALUES (1, '카라멜/치토스 팝콘');
```

```
-- popcorn_order_id가 1인 주문 조회
SELECT
    mr.reserver_name,
```

```
    po.popcorn_order_id,
    po.popcorn_name
FROM
    popcorn_order_s po
JOIN
    movie_reservation_s mr ON po.reservation_id = mr.reservation_id
WHERE
    po.popcorn_order_id = 1;
```

[실행 결과]

reserver_name	popcorn_order_id	popcorn_name
네이트	1	카라멜/치토스 팝콘

조인 조건이 `ON po.reservation_id = mrs.reservation_id` 하나로 매우 **단순하고 명확**해졌다. 이는 쿼리 가독성을 높이고, 개발자가 실수할 가능성을 줄여준다.

이점

- **불변성 확보**: `reservation_id`는 절대 변하지 않으므로, 비즈니스적인 변경(예: 상영 시간 지연, 영화 제목 수정)이 발생해도 키는 안전하다. 다른 테이블에 전파될 영향이 없다.
- **단순한 외래 키 참조**: 다른 테이블에서 단일 컬럼(`reservation_id`)만 참조하면 되므로, 테이블 구조가 단순해지고 저장 공간을 절약할 수 있다.
- **성능 향상**: 크기가 작고 타입이 일관된 숫자 키(`BIGINT`)는 여러 컬럼이 조합된 복합키보다 인덱싱과 조인 연산에서 훨씬 빠르다.
- **유연성 및 명확성**: 자연 키에 해당하는 `{영화 제목, 상영 시간, 좌석 번호}` 조합의 유일성은 `UNIQUE` 제약조건을 통해 비즈니스 규칙으로 계속 강제할 수 있다. 물리적인 기본 키의 책임과 비즈니스 식별자의 책임을 분리하여 모델이 더 명확하고 유연해진다.

복합키 정리

자연 키를 기본 키로 사용하려면, 비즈니스 로직상 하나의 컬럼만으로는 부족하여 여러 컬럼을 묶은 **복합키**를 사용해야 하는 경우가 많다. 복합키는 데이터를 식별하고 다른 테이블과 관계를 맺을 때 **여러 컬럼을 한 묶음**으로 다뤄야 하므로 복잡하고 비효율적이다.

반면 **대리 키**는 비즈니스와 무관한 **단 하나의 컬럼**으로 데이터를 고유하게 식별한다. 덕분에 다른 테이블과의 관계 역시 이 단일 컬럼 하나로 매우 단순하고 명확하게 표현할 수 있다.

이러한 명백한 장점 때문에, 현대적인 데이터베이스 설계에서는 다음의 조합이 사실상의 표준으로 자리 잡았다.

1. **기본 키 (PK)**: **대리 키**를 사용한다.
    - 예: `reservation_id`
2. **비즈니스 제약**: **자연 키**는 `UNIQUE` 제약조건으로 설정한다.
    - 예: `UNIQUE(movie_title, screening_dt, seat_number)`

이 방식을 통해 데이터 무결성을 확실히 지키면서도, 시스템의 유연성과 확장성을 크게 높일 수 있다. 실무에서는 고민의 여지 없이 **대리 키 사용을 우선적으로 고려**하는 것이 좋다.

다대다 관계와 복합키

지금까지 우리는 비즈니스 의미를 가진 자연 키를 복합키로 사용하는 경우의 문제점을 살펴보았다. 그 대안으로 대리 키를 사용하면 많은 문제가 해결된다는 것도 확인했다. 그렇다면 자연 키가 아닌, **다른 테이블의 대리 키(외래 키)들을 조합해서 복합키를 만드는 경우**는 어떨까? 이는 실무에서 아주 흔하게 마주치는 상황이며, 특히 다대다(N:M) 관계를 해결할 때 나타난다.

문제 상황: 다대다 관계 모델링

우리의 쇼핑몰에서 '주문'과 '상품'의 관계를 다시 생각해보자.

- 하나의 **주문**에는 여러 개의 **상품**이 포함될 수 있다. (1:N)
- 하나의 **상품**은 여러 **주문**에 포함될 수 있다. (1:N)

이처럼 관계의 양쪽 모두가 다(N)의 관계를 가질 때, 이를 **다대다(N:M) 관계**라고 부른다.

관계형 데이터베이스에서는 테이블 간의 관계를 외래 키로 표현하는데, 다대다 관계는 두 테이블만으로는 직접 표현할 수 없다. `orders` 테이블에 `product_id` 컬럼을 추가하면 한 주문에는 상품 하나만 담을 수 있게 되고, 반대로 `product` 테이블에 `order_id` 컬럼을 넣으면 하나의 상품은 단 하나의 주문에만 속하게 되는 문제가 발생한다.

> **다대다 관계는 개념적 모델링 단계에서 연관 엔티티를 통해 해결할 수 도 있다.**
> 우리는 앞서 개념적 모델링 단계에서 연관 엔티티를 사용해서 문제를 해결했다.

다대다 관계 해소를 위한 연결 테이블

이 문제를 해결하기 위해, 두 테이블의 중간에서 다리 역할을 하는 **연결 테이블(연관 엔티티)**을 만든다. 우리 쇼핑몰 모델에서는 주문 항목(`order_item`) 테이블이 바로 이 역할을 수행한다. 이 연결 테이블을 통해 주문과 상품의 다대다 관계를 두 개의 일대다(1:N) 관계로 풀어낼 수 있다.

- `orders (1) - (N) order_item (N) - (1) product`
  - `orders` (1) : `order_item` (N)
  - `product` (1) : `order_item` (N)

`order_item` 테이블은 `orders` 테이블의 기본 키인 `order_id`와 `product` 테이블의 기본 키인 `product_id`를 외래 키(FK)로 받아서 두 테이블의 관계를 맺어준다.

여기서 핵심 질문이 나온다. `order_item` 테이블의 기본 키(PK)는 무엇으로 해야 할까? 두 가지 선택지가 있다.

- **방법 1**: `{order_id, product_id}`를 복합 기본 키로 사용한다.
- **방법 2**: `order_item_id`라는 별도의 대리 키를 기본 키로 사용하고, `{order_id, product_id}`에는 UNIQUE 제약조건을 건다.

방법 1: `{order_id, product_id}`를 복합 기본 키로 사용하기

이 방법은 가장 직관적이고 또 전통적인 다대다 관계 모델링에서 선택하는 방식이다. `order_item`의 한 행은 "어떤 주문에 어떤 상품이 포함되었는가"를 나타낸다. 따라서 `{주문 ID, 상품 ID}`의 조합 자체가 이 행을 식별하는 가장 자연스러운 식별자가 된다.

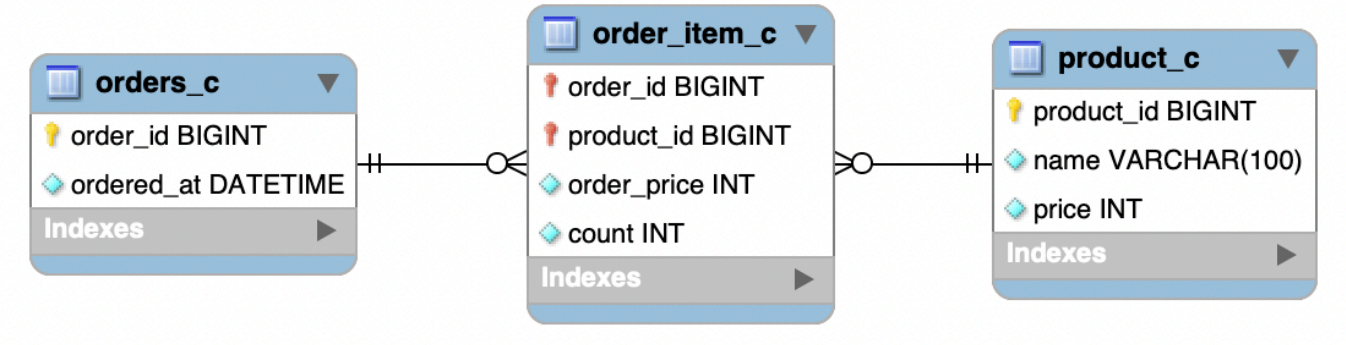
테이블 구조 및 데이터 예시 (SQL)

전체 예제를 실행할 수 있도록 `product_c`, `orders_c`, `order_item_c` 테이블을 순서대로 생성하고 데이터를 삽입해 보자.

```
-- 테이블 초기화
DROP TABLE IF EXISTS order_item_c;
DROP TABLE IF EXISTS orders_c;
DROP TABLE IF EXISTS product_c;

-- 상품 테이블 생성
```

```sql
CREATE TABLE product_c (
    product_id   BIGINT       NOT NULL AUTO_INCREMENT,
    name         VARCHAR(100) NOT NULL,
    price        INT          NOT NULL,
    PRIMARY KEY (product_id)
);

-- 주문 테이블 생성
CREATE TABLE orders_c (
    order_id   BIGINT   NOT NULL AUTO_INCREMENT,
    ordered_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (order_id)
);

-- 주문 항목 테이블 생성 (복합키 사용)
CREATE TABLE order_item_c (
    order_id     BIGINT NOT NULL, -- 주문 ID (PK, FK)
    product_id   BIGINT NOT NULL, -- 상품 ID (PK, FK)
    order_price  INT    NOT NULL, -- 주문 당시 가격
    count        INT    NOT NULL, -- 주문 수량
    PRIMARY KEY (order_id, product_id), -- 복합 기본 키
    CONSTRAINT fk_order_item_c_orders FOREIGN KEY (order_id)
        REFERENCES orders_c (order_id),
    CONSTRAINT fk_order_item_c_product FOREIGN KEY (product_id)
        REFERENCES product_c (product_id)
);

-- 샘플 데이터 삽입
INSERT INTO product_c (product_id, name, price) VALUES (101, '노트북', 1500000);
INSERT INTO product_c (product_id, name, price) VALUES (102, '마우스', 20000);
INSERT INTO orders_c (order_id) VALUES (1);

-- 1번 주문에 101번 상품 2개 추가
INSERT INTO order_item_c (order_id, product_id, order_price, count)
VALUES (1, 101, 1500000, 2);

-- 1번 주문에 102번 상품 1개 추가
INSERT INTO order_item_c (order_id, product_id, order_price, count)
VALUES (1, 102, 20000, 1);

-- (실패) 1번 주문에 101번 상품을 또 추가하려고 시도
-- INSERT INTO order_item_c (order_id, product_id, order_price, count)
-- VALUES (1, 101, 1500000, 3);
```

- 복합키(Composite Key)를 사용하는 예시이므로 학습 목적으로 테이블을 쉽게 구분하기 위해 뒤에 `_c`를 붙였다.



- 부모의 PK를 자신의 PK + FK로 사용하는 관계를 식별 관계라고 하고 실선을 사용한다. 자세한 내용은 뒤의 식별 관계에서 다룬다.

실패하는 INSERT 문 실행 결과

주석 처리된 마지막 `INSERT` 문의 주석을 풀고 실행하면, 기본 키 중복 오류가 발생한다.

```
Error Code: 1062. Duplicate entry '1-101' for key 'order_item_c.PRIMARY'
```

데이터 확인

성공적으로 입력된 데이터를 확인해보자.

```sql
SELECT * FROM order_item_c;
```

[실행 결과]

order_id	product_id	order_price	count
1	101	1500000	2
1	102	20000	1

이처럼 복합키는 데이터의 정합성을 강력하게 지켜준다.

장점

- **데이터 무결성 보장**: 기본 키 제약조건에 의해 `{order_id, product_id}` 조합의 중복이 원천적으로 차단된다. 즉, 하나의 주문에 동일한 상품이 두 번 이상 들어가는 것을 데이터베이스 레벨에서 막아준다.
- **논리적 명확성**: 테이블의 구조만 봐도 "이 테이블은 주문과 상품의 관계를 나타내는구나"라는 것을 명확히 알 수 있다.

하지만 이 방식에도 여전히 우리가 앞서 살펴본 복합키의 단점이 일부 남아있다.

단점

- **PK가 너무 '뚱뚱하다' (Fat Key)**: 기본 키가 두 개의 `BIGINT` 컬럼으로 구성되어 '뚱뚱하다'. 만약 이 `order_item_c` 테이블을 다른 테이블이 참조해야 하는 상황(예: 특정 주문 항목에 대한 문의사항을 저장하는 테이블)이 생긴다면, 외래 키로 이 두 개의 컬럼(`{order_id, product_id}`)을 모두 가져가야 한다. 이는 외래 키 참조를 복잡하게 만들고 저장 공간을 낭비한다.
- **확장성의 제약**: `order_item_c` 테이블 자체가 더 복잡한 관계의 부모 테이블이 될수록 이 '뚱뚱한' 복합키는 계속해서 하위 테이블로 전파되어 모델의 유연성을 떨어뜨린다.
- **ORM 사용의 불편함**: JPA와 같은 현대적인 ORM 프레임워크에서 복합키를 기본 키로 매핑하려면 별도의 식별자 클래스(`@IdClass` 또는 `@EmbeddedId`)를 만들어야 하는 등 개발의 복잡성이 증가한다. 단일 대리 키를 사용하는 것이 훨씬 편리하다.

그래서 현대적인 데이터베이스 설계에서는 다대다 관계를 위한 연결 테이블에서도 **별도의 대리 키를 기본 키로 사용하는 것을 권장한다.**

방법 2: 대리 키 - PK + 복합 UNIQUE 제약조건 사용하기 (권장)

이 방법은 현대적인 데이터베이스 설계에서 더 선호되는 방식이다. `order_item` 테이블 자체의 식별을 위한 대리 키(`order_item_id`)를 두고, 비즈니스적 유일성 보장은 `{order_id, product_id}`에 `UNIQUE` 제약조건을 걸어서 해결한다. 이것이 바로 우리가 최종적으로 채택한 `order_item` 테이블의 구조다.

테이블 구조 및 데이터 예시 (SQL)

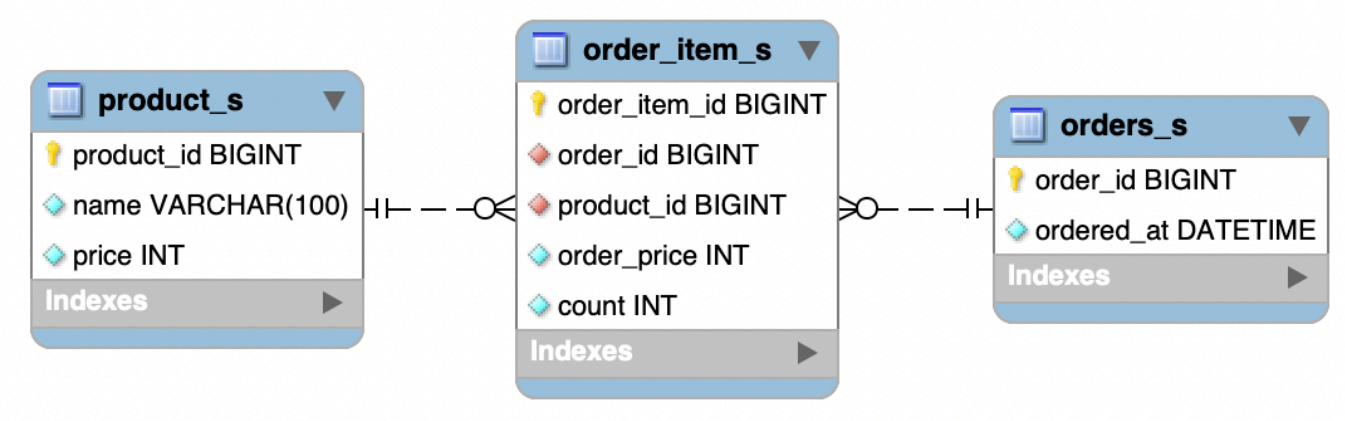
이번에는 대리 키를 사용하는 `_s` 테이블들로 전체 예제를 다시 구성해보자.

```
-- 테이블 초기화
DROP TABLE IF EXISTS order_item_s;
DROP TABLE IF EXISTS orders_s;
DROP TABLE IF EXISTS product_s;

-- 상품 테이블 생성
```

```sql
CREATE TABLE product_s (
    product_id   BIGINT       NOT NULL AUTO_INCREMENT,
    name         VARCHAR(100) NOT NULL,
    price        INT          NOT NULL,
    PRIMARY KEY (product_id)
);

-- 주문 테이블 생성
CREATE TABLE orders_s (
    order_id   BIGINT   NOT NULL AUTO_INCREMENT,
    ordered_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (order_id)
);

-- 주문 항목 테이블 생성 (대리 키 + UNIQUE 제약조건 사용)
CREATE TABLE order_item_s (
    order_item_id BIGINT NOT NULL AUTO_INCREMENT, -- 주문 상품 ID (PK)
    order_id      BIGINT NOT NULL,                -- 주문 ID (FK)
    product_id    BIGINT NOT NULL,                -- 상품 ID (FK)
    order_price   INT    NOT NULL,                -- 주문 당시 가격
    count         INT    NOT NULL,                -- 주문 수량
    PRIMARY KEY (order_item_id),
    CONSTRAINT fk_order_item_s_orders FOREIGN KEY (order_id)
        REFERENCES orders_s (order_id),
    CONSTRAINT fk_order_item_s_product FOREIGN KEY (product_id)
        REFERENCES product_s (product_id),
    UNIQUE KEY uq_order_product (order_id, product_id) -- 복합 UNIQUE 제약
);

-- 샘플 데이터 삽입 (예시를 위해 ID 직접 지정)
INSERT INTO product_s (product_id, name, price) VALUES (101, '노트북', 1500000);
INSERT INTO product_s (product_id, name, price) VALUES (102, '마우스', 20000);
INSERT INTO orders_s (order_id) VALUES (1);

-- 1번 주문에 101번 상품 2개 추가
INSERT INTO order_item_s (order_id, product_id, order_price, count)
VALUES (1, 101, 1500000, 2);

-- 1번 주문에 102번 상품 1개 추가
INSERT INTO order_item_s (order_id, product_id, order_price, count)
VALUES (1, 102, 20000, 1);

-- (실패) 1번 주문에 101번 상품을 중복 추가 시도
```

```
-- INSERT INTO order_item_s (order_id, product_id, order_price, count)
-- VALUES (1, 101, 1500000, 3);
```

- 대리 키(Surrogate Key)를 사용하는 예시이므로 학습 목적으로 테이블을 쉽게 구분하기 위해 뒤에 `_s`를 붙였다.



실패하는 INSERT 문 실행 결과

마찬가지로 마지막 `INSERT` 문을 실행하면 `UNIQUE` 제약조건 위반으로 오류가 발생한다.

```
Error Code: 1062. Duplicate entry '1-101' for key
'order_item_s.uq_order_product'
```

오류 메시지에서 'PRIMARY' 키가 아닌 `uq_order_product`라는 `UNIQUE` 키에서 중복이 발생했음을 알려준다.
기능적으로는 복합 기본 키와 동일한 역할을 수행하는 것을 알 수 있다.

데이터 확인

```
SELECT * FROM order_item_s;
```

[실행 결과]

order_item_id	order_id	product_id	order_price	count
1	1	101	1500000	2
2	1	102	20000	1

이 구조의 핵심은 다음과 같다.

1. **기본 키는 단순하게**: 비즈니스와 무관한 `order_item_id`(대리 키)를 기본 키(PK)로 사용한다. 이 PK는 **불변**하며, 작고, 다른 테이블에서 참조하기 매우 용이하다.
2. **비즈니스 제약은 `UNIQUE` 로**: '하나의 주문에 동일한 상품이 중복될 수 없다'는 중요한 비즈니스 규칙은 `order_id`와 `product_id`를 묶은 **복합 `UNIQUE` 제약조건**(`uq_order_product`)을 통해 완벽하게 보장한다.

장점

- **유연하고 일관된 참조**: 다른 테이블이 `order_item_s`의 특정 행을 참조할 때, `order_item_id`라는 단일 컬럼만 외래 키로 사용하면 되므로 구조가 단순하고 명확해진다. (예: 특정 주문 항목에 대한 문의사항을 저장하는 테이블)
- **데이터 무결성 유지**: `UNIQUE` 제약조건이 복합키와 동일한 역할을 수행하여 `{order_id, product_id}` 조합의 중복을 막아준다.
- **개발 편의성**: `order_item_id`라는 단순한 기본 키가 있으므로, JPA 같은 ORM에서 다루기가 매우 쉽다. 복잡한 식별자 클래스 없이 일반적인 엔티티처럼 편리하게 관리할 수 있다.

단점

- **약간의 오버헤드**: 기본 키 인덱스와 `UNIQUE` 인덱스가 별도로 생성되므로 약간의 저장 공간 및 인덱스 관리 비용이 추가될 수 있다. 하지만 현대 하드웨어 환경에서는 거의 무시할 수 있는 수준이다.

실무 기반 데이터베이스 키 선택 전략

데이터베이스 설계 시, 다대다(N:M) 관계에서도 **대리 키(Surrogate Key)를 기본 키(PK)로 사용하고, 비즈니스 키 컬럼들을 복합 유니크(UNIQUE) 제약 조건으로 설정하는 방식이 가장 실용적이고 현대적인 해결책**이다. 이 접근법은 복합키의 **데이터 정합성 보장**이라는 장점과 대리 키의 **단순성, 불변성, 확장성**이라는 장점을 모두 취할 수 있다.

애플리케이션의 비즈니스 규칙은 변할 수 있지만, 데이터베이스의 구조적 안정성은 유지되어야 한다. 대리 키는 바로 이 **안정적인 뼈대**를 제공하여 다음과 같은 이점을 가져다준다.

- **개발 복잡도 감소:** 복잡한 복합키 대신 단순한 대리 키를 사용하여 테이블 간의 관계(Join)를 단순화하고 개발 편의성을 높인다.
- **유연성 및 확장성 확보:** 향후 비즈니스 로직 변경이나 컬럼 추가 시, 기본 키 자체의 변경 없이 유연하게 대응할 수 있다.
- **유지보수 용이성:** 일관되고 예측 가능한 구조는 장기적으로 시스템을 유지보수하기 쉽게 만든다.

이러한 이유로, `(order_id, product_id)`라는 훌륭한 복합키가 존재함에도 불구하고, 현대 데이터베이스 설계에서는 `order_item_id`와 같은 별도의 **단일 대리 키**를 기본 키로 추가하는 설계를 더 선호한다. 이것이 유지보수와 확장에 더 유리한 선택이다.

결론적으로, 데이터베이스 모델링은 애플리케이션의 전체 구조와 직결되므로, 안정성과 유연성을 고려할 때 **대리 키를 기본 키로 사용하는 것이 현대 개발 환경의 표준적인 선택**이라 할 수 있다.

이제 우리는 데이터의 유일성을 보장하고 관계를 맺는 다양한 '키'들에 대해 배웠다. 다음 시간에는 이 키들을 활용하여 테이블 간의 다양한 관계(1:1, 1:N, N:1, N:M, 식별/비식별)를 어떻게 논리적으로 설계하는지 더 깊이 있게 알아보겠다.

정리

다양한 종류의 키

- 데이터베이스 설계는 **개념적, 논리적, 물리적 모델링**의 3단계로 진행한다.
- **논리적 모델링**은 개념 모델을 관계형 데이터베이스 구조에 맞게 변환하는 과정이며, **키(Key)** 정의가 핵심이다.
- 키는 각 행(Row)을 유일하게 식별하고, 테이블 간의 관계를 맺으며, 데이터 무결성을 보장하는 핵심 장치이다.
- **기본 키(Primary Key)**: 테이블의 대표 키로, NULL이 없고 유일하며 변하지 않아야 한다.
- **후보 키(Candidate Key)**: 기본 키가 될 수 있는 키로, 유일성과 최소성을 만족한다.
- **외래 키(Foreign Key)**: 다른 테이블의 기본 키를 참조하여 테이블 간의 관계를 연결하는 역할을 한다.

자연 키 vs 대리 키1 - 자연 키

- **자연 키(Natural Key)**는 주민등록번호, 이메일처럼 비즈니스 로직에서 자연스럽게 발생하는 의미 있는 데이터를 기본 키로 사용하는 것이다.
- 자연 키의 치명적 약점은 **변경 가능성**이다. 실무에서 기본 키는 절대로 변하면 안 된다.
- 자연 키를 기본 키로 사용하면, 해당 값이 변경될 때 참조 무결성 제약조건 위반, 연쇄 업데이트로 인한 시스템 부하, 데이터 역사성 훼손, 외부 시스템 연동 문제 등 심각한 문제가 발생할 수 있다.

자연 키 vs 대리 키2 - 대리 키

- **대리 키(Surrogate Key)**는 비즈니스 로직과 무관한, 오직 데이터 식별을 위해 시스템이 자동 생성하는 값(예: 1, 2, 3... 또는 UUID)을 기본 키로 사용하는 것이다.
- 대리 키는 절대 변하지 않으므로, 비즈니스 로직(예: 이메일 주소)이 변경되어도 데이터 구조의 안정성이 유지된다.

- **현대 데이터베이스 설계의 표준 패턴**은 다음과 같다.
  - **대리 키**를 **기본 키(PK)**로 사용하여 관계의 안정성을 확보한다.
  - **자연 키**에는 **UNIQUE 제약조건**을 적용하여 비즈니스 데이터의 고유성을 보장한다.

자연 키 vs 대리 키3 - 성능 트레이드 오프

- **자연 키**는 일부 상황에 JOIN 없이 조회가 가능해 단순 조회에 유리할 수 있지만, 외래 키의 크기가 커져 공간을 낭비하고 인덱스 단편화로 쓰기 성능을 저하시키는 단점이 있다.
- **대리 키**는 추가적인 JOIN이 발생할 수 있으나, 외래 키 크기가 작아 효율적이고 순차적으로 값이 증가하여 쓰기 성능이 매우 뛰어나다.
- 성능 관점에서도 데이터 모델의 안정성, 유연성, 쓰기 성능의 이점이 더 크기 때문에 대부분 대리 키를 기본 키로 선택한다.

자연 키 vs 대리 키4 - 현대적인 설계

- 현대적인 설계에서는 **대리 키 사용을 강력하게 권장**한다. 이는 데이터의 식별자(ID)를 비즈니스 로직으로부터 완전히 분리하여 **느슨한 결합(Loose Coupling)**을 유지하기 위함이다.
- 비즈니스 요구사항의 빠른 변화, JPA 같은 ORM 기술의 보편화, 마이크로서비스 아키텍처의 등장으로 인해 불변성을 가진 대리 키의 중요성이 더욱 커졌다.

복합키 설계

- **복합키(Composite Key)**는 두 개 이상의 컬럼을 묶어서 하나의 기본 키로 사용하는 것이다.
- 자연 키를 복합키로 사용하면 변경 가능성, 외래 키 참조의 복잡성 및 크기 증가, ORM 사용의 어려움 등 여러 문제가 발생한다.
- 복합키 문제의 대안 역시 **대리 키**를 기본 키(PK)로 사용하고, 비즈니스적 유일성이 필요한 복합 컬럼들에는 **UNIQUE 제약조건**을 설정하는 것이다.

다대다 관계와 복합키

- 다대다(N:M) 관계는 두 테이블을 직접 연결할 수 없어, 중간에 **연결 테이블**을 만들어 해소한다.
- 이 연결 테이블의 기본 키는 두 테이블의 외래 키를 묶은 **복합키**로 만들 수 있다. 이는 데이터 무결성을 보장하지만 복합키의 단점을 그대로 가진다.
- 권장되는 현대적인 방식은 연결 테이블에도 별도의 **대리 키(PK)**를 만들고, 두 외래 키의 조합에는 **복합 UNIQUE 제약조건**을 설정하는 것이다.
- 이 방식은 대리 키의 단순성, 불변성, 확장성과 복합키의 데이터 정합성 보장이라는 장점을 모두 취할 수 있다.

> **대리 키-PK, 자연 키-UNIQUE**
> 현대적인 데이터베이스 설계는 대리 키-PK, 자연 키-UNIQUE 방식이 사실상 표준이다.
> 대리 키-PK, 자연 키-UNIQUE 이것 딱 하나만 기억하자.