8. 상속 관계 설계

#0.강의/2.데이터베이스로드맵/4.설계2

상속 관계 설계 - 문제 상황

객체지향 프로그래밍에서는 상속이라는 개념이 있다. 부모 클래스의 속성과 행동을 자식 클래스가 물려받는 것이다. 그런데 관계형 데이터베이스에는 상속이라는 개념이 없다. 그렇다면 상속 구조를 데이터베이스에서는 어떻게 표현해야 할까?

물론 PostgreSQL 같은 일부 DB가 상속 기능을 지원하긴 하지만, 우리는 표준적이고 범용적인 설계를 배워야 한다. 일반적인 RDBMS는 테이블과 관계(FK)만 있을 뿐, 상속 기능은 없다.

그렇다면 객체의 상속 구조를 어떻게 테이블로 옮길 것인가? 이것이 오늘 우리가 해결해야 할 핵심 과제다.
이러한 논리적 설계를 데이터 모델링에서는 **슈퍼타입-서브타입(Super-Type/Sub-Type) 모델**이라고 부른다.

이번 시간에는 상속 관계를 데이터베이스 테이블로 설계하는 다양한 전략을 알아보겠다.

문제 상황 - 쇼핑몰의 다양한 상품

우리 쇼핑몰이 성장하면서 다양한 종류의 상품을 판매하게 되었다. 처음에는 도서만 팔았는데, 이제는 전자제품과 의류도 함께 판매한다.

각 상품 유형별로 필요한 정보를 정리해보자.

공통 속성 (모든 상품)

- 상품ID
- 상품명
- 가격
- 재고수량
- 등록일

도서(Book)만의 속성

- 저자
- ISBN
- 출판사

전자제품(Electronics)만의 속성

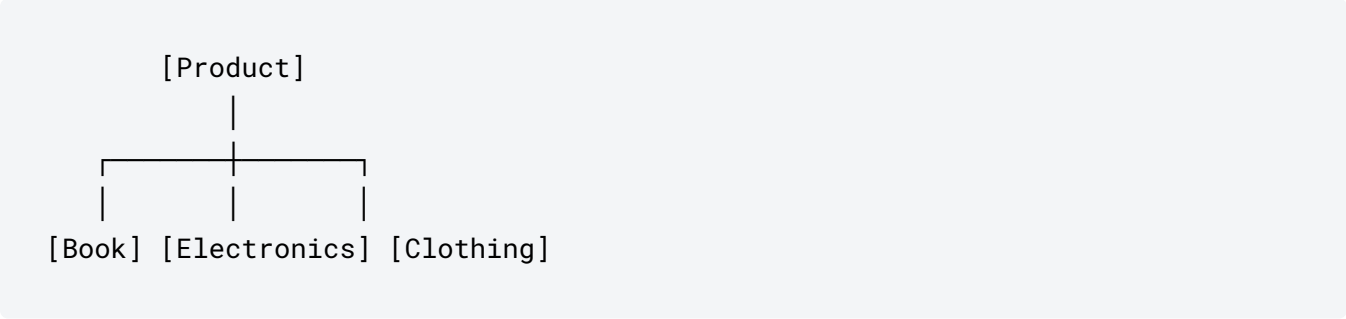
- 제조사
- 보증기간(개월)
- 전력소비량(W)

의류(Clothing)만의 속성

- 사이즈
- 색상
- 소재

도서, 전자제품, 의류 모두 상품명, 가격, 재고수량, 등록일 같은 공통 속성을 가지는 것이다.

이것을 객체지향적으로 표현하면 다음과 같은 상속 구조가 된다.



`Product`라는 부모가 있고, `Book`, `Electronics`, `Clothing`이라는 자식들이 있다. 자식들은 부모의 공통 속성을 물려받고, 각자 고유한 속성을 추가로 가진다.

문제는 관계형 데이터베이스에는 이런 상속 개념이 없다는 것이다. 그렇다면 이 구조를 테이블로 어떻게 표현해야 할

까? 단순 무식하게 각자 테이블을 만들고 공통 속성을 각자 가지면 될까? 이렇게 하면 공통 속성이 중복되기 때문에 향후 공통 속성을 추가하거나 변경할 때 모든 테이블을 다 변경해야 하는 문제가 발생한다.

우리는 이 문제를 해결하기 위해 총 3가지 전략을 순서대로 살펴볼 것이다.

1. 구현 클래스마다 테이블 전략 (Table-per-Concrete-Class)
2. 단일 테이블 전략 (Single-Table)
3. 조인 전략 (Joined-Table)

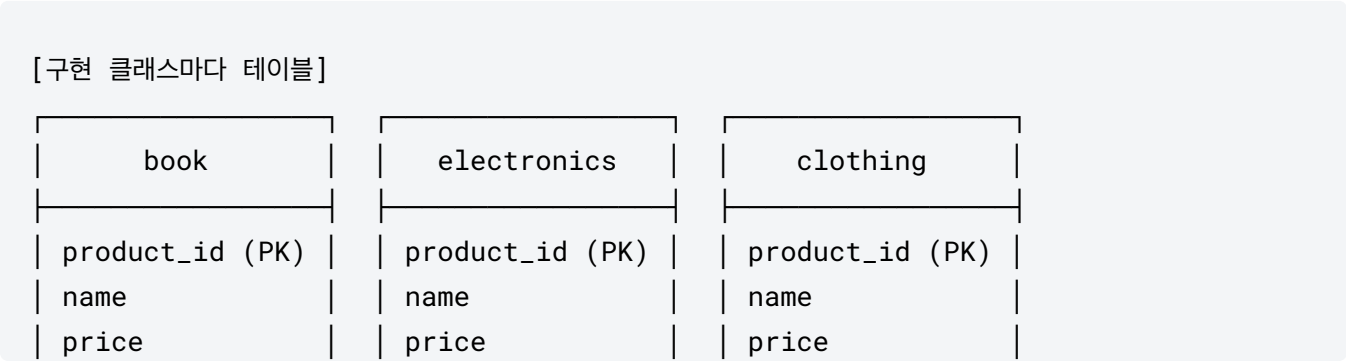
각 전략을 하나씩 살펴보면서 장단점을 비교해보겠다.

> **용어 사용**
> 강의에서는 이해하기 쉽고 간단하게 개발자들이 많이 사용하는 용어를 선택하겠다.
> 관계형 데이터베이스 모델링 용어는 오른쪽 화살표를 참고하자.
>
> 구현 클래스마다 테이블 전략 ➡ 서브타입별 개별 테이블 전략 (Roll-down)
> 단일 테이블 전략 ➡ 슈퍼타입 통합 테이블 전략 (Roll-up)
> 조인 전략 ➡ 슈퍼/서브타입 개별 테이블 전략 (One-to-One)

구현 클래스마다 테이블 전략

먼저 가장 무식한 방법부터 살펴보자.
"구현 클래스마다 테이블 전략"은 자식 클래스마다 별도의 테이블을 만들고, 각 테이블에 부모의 속성까지 모두 포함시키는 방식이다.

테이블 설계



```
| author          |   | manufacturer    |   | size            |
| isbn            |   | warranty_months |   | color           |
└─────────────────┘   └─────────────────┘   └─────────────────┘
```

- 일부 컬럼 생략

```sql
DROP TABLE IF EXISTS book;
DROP TABLE IF EXISTS electronics;
DROP TABLE IF EXISTS clothing;

-- 도서 테이블 (부모 속성 + 도서 고유 속성)
CREATE TABLE book (
    product_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(200) NOT NULL,
    price INT NOT NULL,
    stock_quantity INT NOT NULL DEFAULT 0,
    created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    -- 도서 고유 속성
    author VARCHAR(100) NOT NULL,
    isbn VARCHAR(13),
    publisher VARCHAR(100)
);

-- 전자제품 테이블 (부모 속성 + 전자제품 고유 속성)
CREATE TABLE electronics (
    product_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(200) NOT NULL,
    price INT NOT NULL,
    stock_quantity INT NOT NULL DEFAULT 0,
    created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    -- 전자제품 고유 속성
    manufacturer VARCHAR(100) NOT NULL,
    warranty_months INT NOT NULL DEFAULT 12,
    power_consumption INT
);

-- 의류 테이블 (부모 속성 + 의류 고유 속성)
CREATE TABLE clothing (
    product_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(200) NOT NULL,
    price INT NOT NULL,
    stock_quantity INT NOT NULL DEFAULT 0,
```

```
    created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    -- 의류 고유 속성
    size VARCHAR(10) NOT NULL,
    color VARCHAR(50) NOT NULL,
    material VARCHAR(100)
);
```

각 테이블이 `name`, `price`, `stock_quantity`, `created_at`이라는 공통 속성을 모두 가지고 있다. 그리고 각자 고유한 속성들을 추가로 가진다.

데이터 입력

```
-- 도서 데이터 입력
INSERT INTO book (name, price, stock_quantity, author, isbn, publisher)
VALUES
    ('클린 코드', 33000, 100, '로버트 마틴', '9788966260959', '인사이트'),
    ('이펙티브 자바', 36000, 50, '조슈아 블로크', '9788966262281', '인사이트'),
    ('JPA 프로그래밍', 43000, 30, '김영한', '9788960777330', '에이콘');

-- 전자제품 데이터 입력
INSERT INTO electronics (name, price, stock_quantity, manufacturer,
warranty_months, power_consumption)
VALUES
    ('갤럭시', 1800000, 200, '삼성전자', 24, 15),
    ('맥북 프로 14', 2500000, 50, '애플', 12, 96),
    ('LG 그램 17', 1800000, 80, 'LG전자', 24, 65);

-- 의류 데이터 입력
INSERT INTO clothing (name, price, stock_quantity, size, color, material)
VALUES
    ('오버핏 맨투맨', 35000, 500, 'L', '블랙', '면 100%'),
    ('슬림핏 청바지', 59000, 300, 'M', '인디고', '데님'),
    ('패딩 점퍼', 189000, 100, 'XL', '네이비', '폴리에스터');
```

조회하기

각 상품 유형별로 조회하는 것은 간단하다.

```sql
-- 도서 조회
SELECT product_id, name, price, stock_quantity, author, publisher
FROM book;
```

[실행 결과]

product_id	name	price	stock_quantity	author	publisher
1	클린 코드	33000	100	로버트 마틴	인사이트
2	이펙티브 자바	36000	50	조슈아 블로크	인사이트
3	JPA 프로그래밍	43000	30	김영한	에이콘

```sql
-- 전자제품 조회
SELECT product_id, name, price, manufacturer, warranty_months
FROM electronics;
```

[실행 결과]

product_id	name	price	manufacturer	warranty_months
1	갤럭시	1800000	삼성전자	24
2	맥북 프로 14	2500000	애플	12
3	LG 그램 17	1800000	LG전자	24

구현 클래스마다 테이블 전략의 장단점

구현 클래스마다 테이블 전략의 장점

단순하고 직관적이다

테이블 구조가 명확하다. 도서는 도서 테이블, 전자제품은 전자제품 테이블에서 찾으면 된다. 조인 없이 단일 테이블만 조회하면 되므로 쿼리가 단순하다.

각 유형별 조회 성능이 좋다

특정 유형의 상품만 조회할 때 해당 테이블만 보면 된다. 다른 유형의 데이터가 섞여있지 않으므로 특정 유형의 상품만 조회할 때는 조회 성능이 좋다.

NOT NULL 제약조건을 걸 수 있다

각 테이블에 해당 유형에 필요한 컬럼만 있으므로, `NOT NULL` 제약조건을 자유롭게 설정할 수 있다.

구현 클래스마다 테이블 전략의 단점

하지만 이 전략에는 심각한 문제들이 있다. 실무에서 이 전략을 사용하기 어려운 이유를 살펴보자.

문제점 - 전체 상품 조회가 어렵다

쇼핑몰에서 "전체 상품 목록"을 보여줘야 하는 상황을 생각해보자. 메인 페이지에서 신상품을 보여주거나, 검색 결과에서 모든 종류의 상품을 함께 보여줘야 한다.

```sql
-- 전체 상품을 최신순으로 조회하려면?
SELECT product_id, name, price, created_at, 'BOOK' AS product_type
FROM book
UNION ALL
SELECT product_id, name, price, created_at, 'ELECTRONICS' AS product_type
FROM electronics
UNION ALL
SELECT product_id, name, price, created_at, 'CLOTHING' AS product_type
FROM clothing
ORDER BY created_at DESC;
```

[실행 결과]

product_id	name	price	created_at	product_type
3	패딩 점퍼	189000	2026-01-15 10:03:00	CLOTHING
3	LG 그램 17	1800000	2026-01-15 10:02:00	ELECTRONICS

3	JPA 프로그래밍	43000	2026-01-15 10:01:00	BOOK
2	슬림핏 청바지	59000	2026-01-15 10:00:30	CLOTHING
...	...	...	...	...

모든 테이블을 `UNION ALL`로 합쳐야 한다. 상품 유형이 늘어날 때마다 `UNION`을 추가해야 한다. 만약 가구, 식품, 스포츠용품 등이 추가된다면? 쿼리가 점점 복잡해지고 유지보수가 어려워진다.

문제점 - 상품 ID로 조회할 때 어느 테이블인지 모른다

주문 테이블에서 상품 ID를 참조한다고 생각해보자.

```
CREATE TABLE orders (
    order_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    product_id BIGINT NOT NULL,  -- 이 상품이 어느 테이블에 있는지 알 수 없다!
    quantity INT NOT NULL,
    order_date DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP
    ...
);
```

- 실행되지 않는 예시

`product_id`가 1인 상품을 조회하려면 어느 테이블을 봐야 할까? 도서일 수도 있고, 전자제품일 수도 있고, 의류일 수도 있다. 결국 세 테이블을 모두 뒤져봐야 한다.

```
-- product_id = 1인 상품 찾기 (어느 테이블에 있는지 모름)
SELECT 'BOOK' AS product_type, name, price
FROM book WHERE product_id = 1
UNION ALL
SELECT 'ELECTRONICS' AS product_type, name, price
FROM electronics WHERE product_id = 1
UNION ALL
SELECT 'CLOTHING' AS product_type, name, price
FROM clothing WHERE product_id = 1;
```

매우 비효율적이다. 실무에서는 이런 조회가 수없이 발생하는데, 매번 모든 테이블을 조회해야 한다.

문제점 - 외래 키 설정이 불가능하다

더 심각한 문제가 있다. 주문 테이블에서 상품 테이블로 외래 키를 설정할 수 없다.

```
-- 이런 외래 키는 설정할 수 없다!
-- product_id가 book, electronics, clothing 중 어디를 참조해야 하는가?
ALTER TABLE orders
ADD CONSTRAINT fk_orders_product
FOREIGN KEY (product_id) REFERENCES ???(product_id);
```

외래 키는 하나의 테이블만 참조할 수 있다. 여러 테이블 중 하나를 참조하는 것은 불가능하다. 따라서 데이터 무결성을 데이터베이스 레벨에서 보장할 수 없다.

문제점 - 상품 ID 중복 위험

각 테이블이 독립적으로 `AUTO_INCREMENT`를 사용하므로 상품 ID가 중복될 수 있다.

```
-- book 테이블의 product_id = 1 (클린 코드)
-- electronics 테이블의 product_id = 1 (갤럭시)
-- clothing 테이블의 product_id = 1 (오버핏 맨투맨)
```

세 개의 서로 다른 상품이 모두 `product_id = 1`을 가진다. 주문 테이블에서 `product_id = 1`을 참조하면 어떤 상품인지 구분할 수 없다.

이를 해결하려면 별도의 `product_type` 컬럼을 추가해야 한다.

```
CREATE TABLE orders (
    order_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    product_type VARCHAR(20) NOT NULL,  -- 'BOOK', 'ELECTRONICS', 'CLOTHING'
    product_id BIGINT NOT NULL,
    quantity INT NOT NULL,
    order_date DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP
);
```

또는 모든 상품의 키를 통합해서 별도로 관리해야 한다.

하지만 이렇게 해도 테이블이 나누어져 있으므로 외래 키는 설정할 수 없고, 애플리케이션에서 관리해야 하는 복잡성이 증가한다.

문제점 - 컬럼 변경 시 모든 테이블 수정 필요
공통 속성을 변경해야 할 때를 생각해보자. 예를 들어 모든 상품에 `discount_rate`(할인율) 컬럼을 추가하려면?

```
ALTER TABLE book ADD COLUMN discount_rate INT DEFAULT 0;
ALTER TABLE electronics ADD COLUMN discount_rate INT DEFAULT 0;
ALTER TABLE clothing ADD COLUMN discount_rate INT DEFAULT 0;
-- 새로운 상품 유형이 추가될 때마다 수정 필요
```

테이블이 많아질수록 공통 속성의 변경 작업이 번거롭고 누락 위험이 있다.

구현 클래스마다 테이블 전략 정리

장점	단점
구조가 단순하고 직관적	전체 상품 조회 시 UNION 필요
각 유형별 조회 성능 우수	상품 ID로 조회 시 모든 테이블 검색
NOT NULL 제약조건 사용 가능	외래 키 설정 불가능
	상품 ID 중복 위험
	공통 속성 변경 시 모든 테이블 수정

이러한 단점들 때문에 구현 클래스마다 테이블 전략은 실무에서 거의 사용하지 않는다. 특별한 이유가 없다면 피하는 것이 좋다.

단일 테이블 전략

구현 클래스마다 테이블 전략의 가장 큰 문제는 테이블이 분리되어 있다는 것이었다. 그렇다면 모든 데이터를 하나의 테이블에서 관리하면 어떨까?

단일 테이블 전략(Single Table Strategy)은 부모와 모든 자식의 모든 속성을 하나의 테이블에 통합하는 방식이다.

테이블 설계

```
[단일 테이블]

┌──────────────────────────────────────────────────────────┐
│                         product                          │
├──────────────────────────────────────────────────────────┤
│ product_id (PK), product_type, name, price               │
│ author, isbn, publisher (BOOK)                           │
│ manufacturer, warranty_months (ELECTRONICS)              │
│ size, color, material (CLOTHING)                         │
└──────────────────────────────────────────────────────────┘
```

- 일부 컬럼 생략

```sql
-- 기존 테이블 정리
DROP TABLE IF EXISTS book;
DROP TABLE IF EXISTS electronics;
DROP TABLE IF EXISTS clothing;
DROP TABLE IF EXISTS product;

CREATE TABLE product (
    product_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    product_type VARCHAR(20) NOT NULL,  -- 'BOOK', 'ELECTRONICS', 'CLOTHING'

    -- 공통 속성
    name VARCHAR(200) NOT NULL,
    price INT NOT NULL,
    stock_quantity INT NOT NULL DEFAULT 0,
    created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,

    -- 도서 전용 속성
    author VARCHAR(100),
```

```
    isbn VARCHAR(13),
    publisher VARCHAR(100),

    -- 전자제품 전용 속성
    manufacturer VARCHAR(100),
    warranty_months INT,
    power_consumption INT,

    -- 의류 전용 속성
    size VARCHAR(10),
    color VARCHAR(50),
    material VARCHAR(100),

    INDEX idx_product_type (product_type)
);
```

모든 속성이 하나의 테이블에 통합되어 있다. 따라서 각각의 상품을 구분하는 방법이 필요하다.
여기서는 `product_type` 컬럼으로 상품 유형을 구분한다.
참고로 이 컬럼을 "구분자 컬럼(Discriminator Column)"이라고 부른다. 실무에서는 줄여서 `dtype`이라는 컬럼명으로도 사용한다.

데이터 입력

```
-- 도서 데이터 입력
INSERT INTO product (product_type, name, price, stock_quantity, author, isbn,
publisher)
VALUES
    ('BOOK', '클린 코드', 33000, 100, '로버트 마틴', '9788966260959', '인사이트'),
    ('BOOK', '이펙티브 자바', 36000, 50, '조슈아 블로크', '9788966262281', '인사이트'),
    ('BOOK', 'JPA 프로그래밍', 43000, 30, '김영한', '9788960777330', '에이콘');

-- 전자제품 데이터 입력
INSERT INTO product (product_type, name, price, stock_quantity, manufacturer,
warranty_months, power_consumption)
VALUES
    ('ELECTRONICS', '갤럭시', 1800000, 200, '삼성전자', 24, 15),
    ('ELECTRONICS', '맥북 프로 14', 2500000, 50, '애플', 12, 96),
    ('ELECTRONICS', 'LG 그램 17', 1800000, 80, 'LG전자', 24, 65);
```

```
-- 의류 데이터 입력
INSERT INTO product (product_type, name, price, stock_quantity, size, color,
material)
VALUES
    ('CLOTHING', '오버핏 맨투맨', 35000, 500, 'L', '블랙', '면 100%'),
    ('CLOTHING', '슬림핏 청바지', 59000, 300, 'M', '인디고', '데님'),
    ('CLOTHING', '패딩 점퍼', 189000, 100, 'XL', '네이비', '폴리에스터');
```

- 하나의 테이블에 각자의 컬럼에 맞추어 데이터를 저장하면 된다.

전체 상품 조회

이제 전체 상품 조회가 매우 간단해졌다.

```
SELECT product_id, product_type, name, price, stock_quantity
FROM product
ORDER BY product_id DESC;
```

[실행 결과]

product_id	product_type	name	price	stock_quantity
9	CLOTHING	패딩 점퍼	189000	100
8	CLOTHING	슬림핏 청바지	59000	300
7	CLOTHING	오버핏 맨투맨	35000	500
6	ELECTRONICS	LG 그램 17	1800000	80
5	ELECTRONICS	맥북 프로 14	2500000	50
4	ELECTRONICS	갤럭시	1800000	200
3	BOOK	JPA 프로그래밍	43000	30
2	BOOK	이펙티브 자바	36000	50
1	BOOK	클린 코드	33000	100

`UNION ALL` 없이 단일 쿼리로 모든 상품을 조회할 수 있다.

유형별 상품 조회

특정 유형의 상품만 조회하는 것도 간단하다.

```sql
-- 도서만 조회
SELECT product_id, name, price, author, publisher
FROM product
WHERE product_type = 'BOOK';
```

[실행 결과]

product_id	name	price	author	publisher
1	클린 코드	33000	로버트 마틴	인사이트
2	이펙티브 자바	36000	조슈아 블로크	인사이트
3	JPA 프로그래밍	43000	김영한	에이콘

```sql
-- 전자제품만 조회
SELECT product_id, name, price, manufacturer, warranty_months
FROM product
WHERE product_type = 'ELECTRONICS';
```

[실행 결과]

product_id	name	price	manufacturer	warranty_months
4	갤럭시	1800000	삼성전자	24
5	맥북 프로 14	2500000	애플	12
6	LG 그램 17	1800000	LG전자	24

상품 ID로 조회

주문 테이블에서 상품을 참조할 때도 간단해졌다.

```
-- product_id = 5인 상품 조회
SELECT product_id, product_type, name, price
FROM product
WHERE product_id = 5;
```

[실행 결과]

product_id	product_type	name	price
5	ELECTRONICS	맥북 프로 14	2500000

단일 테이블이므로 바로 찾을 수 있다. 여러 테이블을 뒤질 필요가 없다.

외래 키 설정

이제 주문 테이블에서 상품 테이블로 외래 키를 설정할 수 있다.

```
DROP TABLE IF EXISTS orders;
CREATE TABLE orders (
    order_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    product_id BIGINT NOT NULL,
    quantity INT NOT NULL,
    order_date DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (product_id) REFERENCES product(product_id)
);

-- 주문 데이터 입력
INSERT INTO orders (product_id, quantity) VALUES (1, 2);  -- 클린 코드 2권
INSERT INTO orders (product_id, quantity) VALUES (4, 1);  -- 갤럭시 1개
INSERT INTO orders (product_id, quantity) VALUES (7, 3);  -- 오버핏 맨투맨 3개
```

외래 키가 정상적으로 작동하므로 존재하지 않는 상품을 참조하면 오류가 발생한다.

```
-- 존재하지 않는 상품 참조 시도
INSERT INTO orders (product_id, quantity) VALUES (999, 1);
```

[실행 결과]

```
Error Code: 1452. Cannot add or update a child row: a foreign key constraint
fails...
```

데이터 무결성을 데이터베이스 레벨에서 보장할 수 있게 되었다.

주문과 상품 조인

주문 정보와 상품 정보를 함께 조회하는 것도 간단하다.

```
SELECT
    o.order_id,
    o.quantity,
    p.product_type,
    p.name,
    p.price,
    (o.quantity * p.price) AS total_price
FROM orders o
JOIN product p ON o.product_id = p.product_id;
```

[실행 결과]

order_id	quantity	product_type	name	price	total_price
1	2	BOOK	클린 코드	33000	66000
2	1	ELECTRONICS	갤럭시	1800000	1800000
3	3	CLOTHING	오버핏 맨투맨	35000	105000

단일 테이블 전략의 장단점

단일 테이블 전략은 구현 클래스마다 테이블 전략의 많은 문제를 해결했다. 이 전략의 장단점을 알아보자.

단일 테이블 전략의 장점

조회 성능이 우수하다

모든 데이터가 하나의 테이블에 있으므로 UNION, 조인이 필요 없다. 전체 상품 조회, 유형별 조회, 상품 ID로 조회 모두 단일 테이블 접근으로 처리된다.

쿼리가 단순하다

`UNION`이 필요 없고, 복잡한 조인도 없다. SQL이 직관적이고 유지보수하기 쉽다.

외래 키 설정이 가능하다

단일 테이블이므로 다른 테이블에서 외래 키로 참조할 수 있다. 데이터 무결성을 보장할 수 있다.

상품 ID가 유일하다

하나의 테이블에서 `AUTO_INCREMENT`를 사용하므로 상품 ID가 전체에서 유일하다.

공통 속성 변경이 쉽다

공통 속성을 변경하려면 하나의 테이블만 수정하면 된다.

단일 테이블 전략의 단점

NULL 값이 많아진다

가장 큰 단점이다. 도서 데이터에는 전자제품과 의류 관련 컬럼이 모두 NULL이다.

```
SELECT product_id, name, product_type,
       author, manufacturer, size
FROM product;
```

[실행 결과]

product_id	name	product_type	author	manufacturer	size
1	클린 코드	BOOK	로버트 마틴	NULL	NULL
2	이펙티브 자바	BOOK	조슈아 블로크	NULL	NULL
3	JPA 프로그래밍	BOOK	김영한	NULL	NULL
4	갤럭시	ELECTRONICS	NULL	삼성전자	NULL
5	맥북 프로 14	ELECTRONICS	NULL	애플	NULL
6	LG 그램 17	ELECTRONICS	NULL	LG전자	NULL
7	오버핏 맨투맨	CLOTHING	NULL	NULL	L
8	슬림핏 청바지	CLOTHING	NULL	NULL	M
9	패딩 점퍼	CLOTHING	NULL	NULL	XL

여러 종류의 상품이 섞여 있기 때문에, 다른 곳에서 사용하지 않는 값의 경우에는 NULL로 남아있다. 상품 유형이 늘어날수록 NULL 컬럼은 더 많아진다.

NOT NULL 제약조건을 걸 수 없다

공통 속성(이름 가격 등)에는 NOT NULL 제약조건을 걸 수 있다. 하지만 고유 컬럼(상세 속성)에는 NOT NULL 제약조건을 걸 수 없다.

도서의 저자(`author`)는 반드시 있어야 하지만, 테이블 레벨에서 `NOT NULL`을 걸 수 없다. 전자제품이나 의류는 `author`가 NULL이어야 하기 때문이다.

```
-- 이렇게 하면 전자제품, 의류 입력 시 오류 발생
ALTER TABLE product MODIFY author VARCHAR(100) NOT NULL;
```

따라서 공통 속성이 아닌 나머지 컬럼의 데이터 무결성은 애플리케이션 레벨에서 관리해야 한다.

테이블 크기가 커진다

- 모든 컬럼이 하나의 테이블에 있으므로 테이블이 비대해진다.
- 상품 유형이 많아지고 각 유형별 고유 속성이 많아지면 컬럼 수가 급격히 증가한다.

```sql
-- 가구, 식품, 스포츠용품이 추가되면...
ALTER TABLE product ADD COLUMN width INT;           -- 가구
ALTER TABLE product ADD COLUMN height INT;          -- 가구
ALTER TABLE product ADD COLUMN depth INT;           -- 가구
ALTER TABLE product ADD COLUMN expiry_date DATE;    -- 식품
ALTER TABLE product ADD COLUMN calories INT;        -- 식품
ALTER TABLE product ADD COLUMN brand VARCHAR(50);   -- 스포츠용품
-- ... 계속 늘어남
```

상품 유형별 제약 조건 관리가 어렵다

CHECK 제약조건으로 어느 정도 보완할 수 있지만 복잡하다.

```sql
-- 도서일 때는 author가 필수라는 제약조건
ALTER TABLE product ADD CONSTRAINT chk_book_author
CHECK (product_type != 'BOOK' OR author IS NOT NULL);

-- 전자제품일 때는 manufacturer가 필수
ALTER TABLE product ADD CONSTRAINT chk_electronics_manufacturer
CHECK (product_type != 'ELECTRONICS' OR manufacturer IS NOT NULL);

-- 의류일 때는 size가 필수
ALTER TABLE product ADD CONSTRAINT chk_clothing_size
CHECK (product_type != 'CLOTHING' OR size IS NOT NULL);
```

상품 유형이 늘어날 때마다 제약조건도 함께 추가해야 한다.

단일 테이블 전략 정리

장점	단점
조회 성능 우수 (조인 불필요)	NULL 값이 많음
쿼리가 단순함	NOT NULL 제약조건 불가
외래 키 설정 가능	테이블 크기가 커짐

상품 ID가 유일함	유형별 제약 조건 관리 복잡
공통 속성 변경 용이	

단일 테이블 전략을 선택하면 좋은 경우

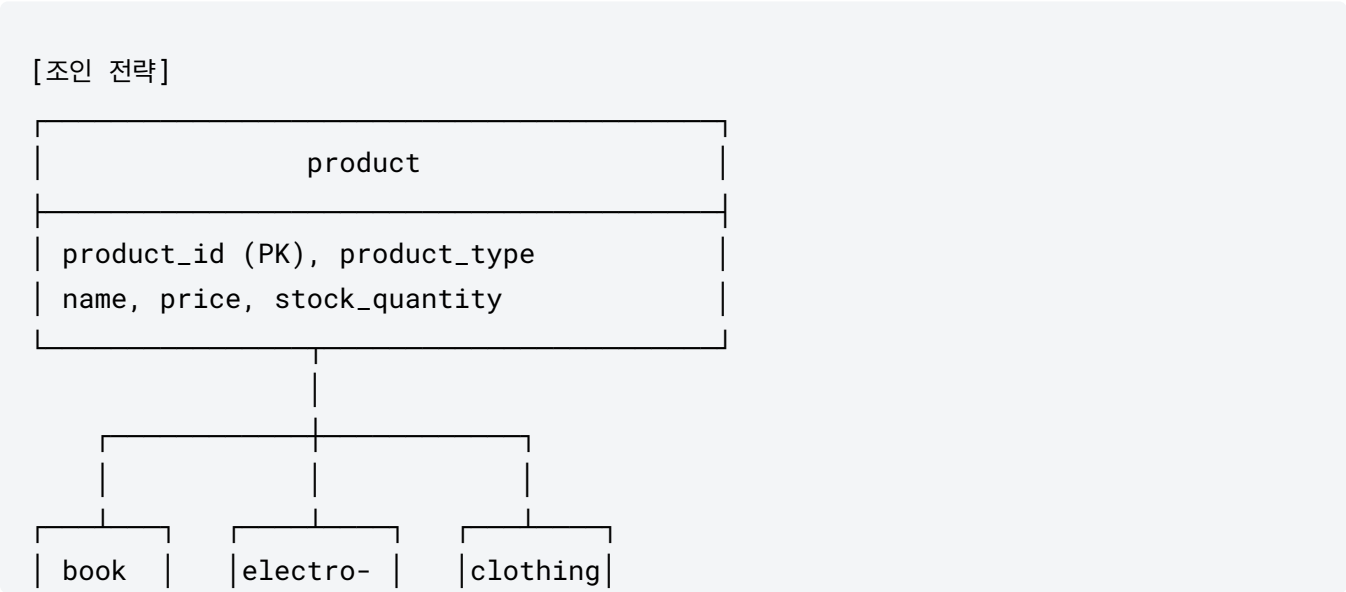
- 분류할 유형의 종류가 적고, 앞으로도 크게 늘어나지 않을 때
- 각 유형별 고유 속성이 적을 때
- 전체 상품 조회가 빈번하고 성능이 중요할 때
- 구조를 단순하게 유지하고 싶을 때

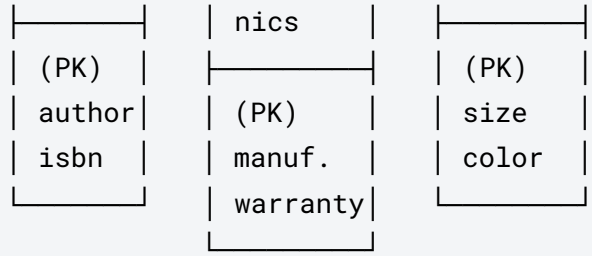
조인 전략

단일 테이블 전략의 가장 큰 문제는 NULL 값이 많다는 것이었다. 이 문제를 해결하면서도 외래 키를 사용할 수 있는 방법이 있을까?

조인 전략(Joined Strategy)은 부모 테이블과 자식 테이블을 분리하고, 조인을 통해 데이터를 조회하는 방식이다. 가장 정석적이고 정규화된 형태라고 볼 수 있다.

테이블 설계





- 일부 컬럼 생략

```
-- 기존 테이블 정리
DROP TABLE IF EXISTS orders;
DROP TABLE IF EXISTS book;
DROP TABLE IF EXISTS electronics;
DROP TABLE IF EXISTS clothing;
DROP TABLE IF EXISTS product;

-- 부모 테이블: 공통 속성을 가진다
CREATE TABLE product (
    product_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    product_type VARCHAR(20) NOT NULL,  -- 구분자 컬럼
    name VARCHAR(200) NOT NULL,
    price INT NOT NULL,
    stock_quantity INT NOT NULL DEFAULT 0,
    created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    INDEX idx_product_type (product_type)
);

-- 자식 테이블: 도서
CREATE TABLE book (
    product_id BIGINT PRIMARY KEY,
    author VARCHAR(100) NOT NULL,
    isbn VARCHAR(13),
    publisher VARCHAR(100),
    FOREIGN KEY (product_id) REFERENCES product(product_id)
);

-- 자식 테이블: 전자제품
CREATE TABLE electronics (
    product_id BIGINT PRIMARY KEY,
    manufacturer VARCHAR(100) NOT NULL,
    warranty_months INT NOT NULL DEFAULT 12,
    power_consumption INT,
```

```
    FOREIGN KEY (product_id) REFERENCES product(product_id)
);

-- 자식 테이블: 의류
CREATE TABLE clothing (
    product_id BIGINT PRIMARY KEY,
    size VARCHAR(10) NOT NULL,
    color VARCHAR(50) NOT NULL,
    material VARCHAR(100),
    FOREIGN KEY (product_id) REFERENCES product(product_id)
);
```

- 부모 테이블인 `product`에는 공통 속성만 있다. 자식 테이블인 `book`, `electronics`, `clothing`에는 각각의 고유 속성만 있다.
- 부모 테이블의 기본 키와 자식 테이블의 기본 키가 같다. **자식 테이블의 기본 키는 부모 테이블의 기본 키를 참조하는 외래 키**이기도 하다.
- 부모와 자식은 PK를 공유하는 1:1 관계이다.

데이터 입력

데이터를 입력할 때는 부모 테이블에 먼저 입력하고, 자식 테이블에 추가 정보를 입력한다.
이 설계에서 주의할 점은 **자식 테이블의 기본 키는 반드시 부모 테이블의 기본 키를 이어받아 사용**해야 한다는 점이다.

```
-- 도서 데이터 입력
INSERT INTO product (product_type, name, price, stock_quantity)
VALUES ('BOOK', '클린 코드', 33000, 100);
INSERT INTO book (product_id, author, isbn, publisher)
VALUES (LAST_INSERT_ID(), '로버트 마틴', '9788966260959', '인사이트');

INSERT INTO product (product_type, name, price, stock_quantity)
VALUES ('BOOK', '이펙티브 자바', 36000, 50);
INSERT INTO book (product_id, author, isbn, publisher)
VALUES (LAST_INSERT_ID(), '조슈아 블로크', '9788966262281', '인사이트');

INSERT INTO product (product_type, name, price, stock_quantity)
VALUES ('BOOK', 'JPA 프로그래밍', 43000, 30);
INSERT INTO book (product_id, author, isbn, publisher)
VALUES (LAST_INSERT_ID(), '김영한', '9788960777330', '에이콘');
```

```sql
-- 전자제품 데이터 입력
INSERT INTO product (product_type, name, price, stock_quantity)
VALUES ('ELECTRONICS', '갤럭시', 1800000, 200);
INSERT INTO electronics (product_id, manufacturer, warranty_months,
power_consumption)
VALUES (LAST_INSERT_ID(), '삼성전자', 24, 15);

INSERT INTO product (product_type, name, price, stock_quantity)
VALUES ('ELECTRONICS', '맥북 프로 14', 2500000, 50);
INSERT INTO electronics (product_id, manufacturer, warranty_months,
power_consumption)
VALUES (LAST_INSERT_ID(), '애플', 12, 96);

INSERT INTO product (product_type, name, price, stock_quantity)
VALUES ('ELECTRONICS', 'LG 그램 17', 1800000, 80);
INSERT INTO electronics (product_id, manufacturer, warranty_months,
power_consumption)
VALUES (LAST_INSERT_ID(), 'LG전자', 24, 65);

-- 의류 데이터 입력
INSERT INTO product (product_type, name, price, stock_quantity)
VALUES ('CLOTHING', '오버핏 맨투맨', 35000, 500);
INSERT INTO clothing (product_id, size, color, material)
VALUES (LAST_INSERT_ID(), 'L', '블랙', '면 100%');

INSERT INTO product (product_type, name, price, stock_quantity)
VALUES ('CLOTHING', '슬림핏 청바지', 59000, 300);
INSERT INTO clothing (product_id, size, color, material)
VALUES (LAST_INSERT_ID(), 'M', '인디고', '데님');

INSERT INTO product (product_type, name, price, stock_quantity)
VALUES ('CLOTHING', '패딩 점퍼', 189000, 100);
INSERT INTO clothing (product_id, size, color, material)
VALUES (LAST_INSERT_ID(), 'XL', '네이비', '폴리에스터');
```

`LAST_INSERT_ID()` 함수를 사용해서 방금 입력한 부모의 `product_id`를 자식 테이블에 입력한다.

데이터 확인

각 테이블의 데이터를 확인해보자.

공통 속성 확인

```sql
SELECT * FROM product;
```

[실행 결과]

product_id	product_type	name	price	stock_quantity	created_at
1	BOOK	클린 코드	33000	100	2026-01-1 10:00:00
2	BOOK	이펙티브 자바	36000	50	2026-01-1 10:00:01
3	BOOK	JPA 프로그래밍	43000	30	2026-01-1 10:00:02
4	ELECTRONICS	갤럭시	1800000	200	2026-01-1 10:00:03
5	ELECTRONICS	맥북 프로 14	2500000	50	2026-01-1 10:00:04
6	ELECTRONICS	LG 그램 17	1800000	80	2026-01-1 10:00:05
7	CLOTHING	오버핏 맨투맨	35000	500	2026-01-1 10:00:06
8	CLOTHING	슬림핏 청바지	59000	300	2026-01-1 10:00:07
9	CLOTHING	패딩 점퍼	189000	100	2026-01-1 10:00:08

```sql
SELECT * FROM book;
```

[실행 결과]

product_id	author	isbn	publisher
1	로버트 마틴	9788966260959	인사이트
2	조슈아 블로크	9788966262281	인사이트
3	김영한	9788960777330	에이콘

```
SELECT * FROM electronics;
```

[실행 결과]

product_id	manufacturer	warranty_months	power_consumption
4	삼성전자	24	15
5	애플	12	96
6	LG전자	24	65

```
SELECT * FROM clothing;
```

[실행 결과]

product_id	size	color	material
7	L	블랙	면 100%
8	M	인디고	데님
9	XL	네이비	폴리에스터

특징

- 각 테이블에 필요한 데이터만 있다. NULL 값이 없다!
- 부모 테이블의 ID와 자식 테이블의 ID가 일치한다.

전체 상품 조회

전체 상품의 공통 정보만 조회할 때는 부모 테이블만 조회하면 된다.

```
SELECT product_id, product_type, name, price, stock_quantity
FROM product
ORDER BY product_id DESC;
```

[실행 결과]

product_id	product_type	name	price	stock_quantity
9	CLOTHING	패딩 점퍼	189000	100
8	CLOTHING	슬림핏 청바지	59000	300
7	CLOTHING	오버핏 맨투맨	35000	500
6	ELECTRONICS	LG 그램 17	1800000	80
5	ELECTRONICS	맥북 프로 14	2500000	50
4	ELECTRONICS	갤럭시	1800000	200
3	BOOK	JPA 프로그래밍	43000	30
2	BOOK	이펙티브 자바	36000	50
1	BOOK	클린 코드	33000	100

유형별 상세 조회

특정 유형의 상품을 상세 정보와 함께 조회하려면 부모 테이블과 해당 자식 테이블의 조인이 필요하다.

```
-- 도서 상세 조회
```

```sql
SELECT p.product_id, p.name, p.price, p.stock_quantity,
       b.author, b.isbn, b.publisher
FROM product p
JOIN book b ON p.product_id = b.product_id;
```

[실행 결과]

product_id	name	price	stock_quantity	author	isbn	p
1	클린 코드	33000	100	로버트 마틴	9788966260959	인
2	이펙티브 자바	36000	50	조슈아 블로크	9788966262281	인
3	JPA 프로그래밍	43000	30	김영한	9788960777330	에

```sql
-- 전자제품 상세 조회
SELECT p.product_id, p.name, p.price, p.stock_quantity,
       e.manufacturer, e.warranty_months, e.power_consumption
FROM product p
JOIN electronics e ON p.product_id = e.product_id;
```

[실행 결과]

product_id	name	price	stock_quantity	manufacturer	warranty_months	powe_cons
4	갤럭시	1800000	200	삼성전자	24	15
5	맥북 프로 14	2500000	50	애플	12	96
6	LG 그램 17	1800000	80	LG전자	24	65

상품 ID로 상세 조회

특정 상품의 모든 정보를 조회하려면 `product_type`을 확인하고 해당 자식 테이블과 조인해야 한다.

```
-- 상품 ID가 2인 상품 조회 (도서인 경우)
SELECT p.product_id, p.product_type, p.name, p.price,
       b.author, b.isbn, b.publisher
FROM product p
LEFT JOIN book b ON p.product_id = b.product_id
WHERE p.product_id = 2;
```

[실행 결과]

product_id	product_type	name	price	author	isbn	publi
2	BOOK	이펙티브 자바	36000	조슈아 블로크	9788966262281	인사이

자식의 타입을 모르는 경우에는 모든 자식 테이블을 LEFT JOIN으로 연결해서 찾을 수 있다.

```
-- 모든 자식 테이블과 LEFT JOIN
SELECT
    p.product_id, p.product_type, p.name, p.price,
    b.author, b.isbn, b.publisher,
    e.manufacturer, e.warranty_months,
    c.size, c.color
FROM product p
LEFT JOIN book b ON p.product_id = b.product_id
LEFT JOIN electronics e ON p.product_id = e.product_id
LEFT JOIN clothing c ON p.product_id = c.product_id
WHERE p.product_id = 5;
```

[실행 결과]

product_id	product_type	name	price	author	manufacturer	size	c

5	ELECTRONICS	맥북 프로 14	2500000	NULL	애플	NULL	N

- 내용이 너무 길어서 일부 컬럼은 제외했다.

모든 자식 테이블 조인(Join)의 동작 원리

이와 같은 쿼리가 가능한 이유는 데이터베이스 설계 시 부모와 자식 테이블이 **동일한 기본 키(PK)**를 공유하도록 설계했기 때문이다.

이 방식이 작동하는 핵심 원리는 다음과 같다.

1. **PK 공유와 1:1 관계**: `book`, `electronics`, `clothing` 테이블의 PK이자 FK인 `product_id`는 부모 테이블 `product`의 `product_id`와 동일한 값을 가진다. 즉, 논리적으로 하나의 상품 정보가 여러 테이블에 쪼개져 저장된 형태다.
2. **LEFT JOIN의 특성**: `LEFT JOIN`은 조인 조건이 일치하지 않는 경우 데이터를 누락시키는 것이 아니라 `NULL`로 채워서 반환한다.
3. **배타적 관계**: 일반적으로 하나의 상품은 '책'이면서 동시에 '전자제품'일 수 없다. 따라서 특정 `product_id`에 대해 자식 테이블 중 하나에만 데이터가 존재하고, 나머지 자식 테이블과의 조인 결과는 모두 `NULL`이 된다.

결과를 살펴보면 `product` 테이블의 공통 정보와 `electronics` 테이블의 상세 정보(`manufacturer`, `warranty_months`)는 정상적으로 조회되었지만, 해당 상품과 관련이 없는 `book`, `clothing` 테이블의 컬럼(`author`, `size` 등)은 모두 `NULL`로 표시된 것을 확인할 수 있다.

모든 상품 상세 조회

자식을 포함한 모든 컬럼을 한번에 조회해보자.
앞서 사용한 쿼리에서 `product_id` 조건만 제거하면 된다.

```sql
-- 모든 자식 테이블과 LEFT JOIN
SELECT
    p.product_id, p.product_type, p.name, p.price,
    b.author, b.isbn, b.publisher,
    e.manufacturer, e.warranty_months,
    c.size, c.color
FROM product p
LEFT JOIN book b ON p.product_id = b.product_id
```

```
LEFT JOIN electronics e ON p.product_id = e.product_id
LEFT JOIN clothing c ON p.product_id = c.product_id;
```

[실행 결과]

product_id	product_type	name	price	author	manufacturer	warranty_months
1	BOOK	클린 코드	33000	로버트..	NULL	NULL
2	BOOK	이펙티브 자바	36000	조슈아..	NULL	NULL
3	BOOK	JPA 프로그래밍	43000	김영한	NULL	NULL
4	ELECTRONICS	갤럭시	1800000	NULL	삼성전자	24
5	ELECTRONICS	맥북 프로 14	2500000	NULL	애플	12
6	ELECTRONICS	LG 그램 17	1800000	NULL	LG전자	24
7	CLOTHING	오버핏 맨투맨	35000	NULL	NULL	NULL
8	CLOTHING	슬림핏 청바지	59000	NULL	NULL	NULL
9	CLOTHING	패딩 점퍼	189000	NULL	NULL	NULL

이처럼 모든 자식 테이블을 `LEFT JOIN`으로 묶어서 조회하면, 상품의 타입이 무엇인지 미리 알지 못해도 한 번의 쿼리로 필요한 모든 상세 정보를 가져올 수 있다. 애플리케이션 계층에서는 이 결과를 받아 `product_type` 여부를 확인하여 적절한 객체로 매핑하면 된다.

이 방식은 조회 쿼리가 단순해진다는 장점이 있지만, 자식 테이블이 늘어날수록 조인 횟수가 많아져 성능에 영향을 줄 수 있으므로 트래픽이 많은 서비스에서는 주의해서 사용해야 한다.

외래 키와 주문 연동

주문 테이블은 부모 테이블인 `product`를 참조한다.

```
DROP TABLE IF EXISTS orders;
```

```sql
CREATE TABLE orders (
    order_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    product_id BIGINT NOT NULL,
    quantity INT NOT NULL,
    order_date DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (product_id) REFERENCES product(product_id)
);

-- 주문 데이터 입력
INSERT INTO orders (product_id, quantity) VALUES (1, 2);   -- 클린 코드 2권
INSERT INTO orders (product_id, quantity) VALUES (4, 1);   -- 갤럭시 1개
INSERT INTO orders (product_id, quantity) VALUES (7, 3);   -- 오버핏 맨투맨 3개
```

주문 정보와 상품 정보를 함께 조회하는 것도 가능하다.

```sql
SELECT
    o.order_id,
    o.quantity,
    p.product_type,
    p.name,
    p.price,
    (o.quantity * p.price) AS total_price
FROM orders o
JOIN product p ON o.product_id = p.product_id;
```

[실행 결과]

order_id	quantity	product_type	name	price	total_price
1	2	BOOK	클린 코드	33000	66000
2	1	ELECTRONICS	갤럭시	1800000	1800000
3	3	CLOTHING	오버핏 맨투맨	35000	105000

주문과 모든 상품 상세 정보 조회

주문 내역을 조회할 때 단순히 상품명과 가격만 필요한 것이 아니다. 책을 주문했으면 '저자'가 누구인지, 옷을 주문했으

면 '사이즈'가 무엇인지 상세 정보도 함께 알고 싶을 것이다.

앞서 배운 **모든 자식 테이블 조인** 기법을 주문 테이블(`orders`)에 적용하면 이 문제를 해결할 수 있다. `orders` 테이블을 기준으로 `product`를 조인하고, 다시 `product`를 기준으로 모든 자식 테이블(`book`, `electronics`, `clothing`)을 `LEFT JOIN`으로 연결하면 된다.

```
SELECT
    o.order_id,
    p.product_type,
    p.name,
    b.author,
    b.isbn,
    e.manufacturer,
    c.size,
    c.color
FROM orders o
JOIN product p ON o.product_id = p.product_id
LEFT JOIN book b ON p.product_id = b.product_id
LEFT JOIN electronics e ON p.product_id = e.product_id
LEFT JOIN clothing c ON p.product_id = c.product_id
ORDER BY o.order_id;
```

[실행 결과]

order_id	product_type	name	author	isbn	manufacturer	size	col
1	BOOK	클린 코드	로버트 마틴	97...	NULL	NULL	NUI
2	ELECTRONICS	갤럭시	NULL	NULL	삼성전자	NULL	NUI
3	CLOTHING	오버핏 맨투맨	NULL	NULL	NULL	L	블랙

쿼리 분석

- `order_id` **1번**: `BOOK` 타입이므로 `book` 테이블의 `author`(로버트 마틴) 정보가 출력되고, 전자제품이나 의류 관련 컬럼은 `NULL`이다.
- `order_id` **2번**: `ELECTRONICS` 타입이므로 `manufacturer`(삼성전자) 정보가 출력된다.

- `order_id` **3번**: `CLOTHING` 타입이므로 `size`(L)와 `color`(블랙) 정보가 출력된다.

조인 전략의 장단점

조인 전략의 장단점을 정리해보자.

장점

정규화된 구조이다

테이블이 정규화되어 있어 데이터 중복이 없다. 각 테이블에 필요한 컬럼만 있어서 저장 공간을 효율적으로 사용한다.

NULL 값이 없다

자식 테이블에 해당 유형에 필요한 컬럼만 있으므로 NULL이 발생하지 않는다.

NOT NULL 제약조건을 사용할 수 있다

각 자식 테이블에서 해당 유형에 필수인 컬럼에 `NOT NULL` 제약조건을 걸 수 있다. 데이터 무결성을 데이터베이스 레벨에서 보장할 수 있다.

외래 키 설정이 가능하다

부모 테이블을 참조하는 외래 키를 설정할 수 있다. 주문, 장바구니 등 다른 테이블에서 상품을 참조하기 편리하다.

유연한 확장이 가능하다

새로운 상품 유형을 추가할 때 기존 테이블을 수정하지 않고 새 자식 테이블만 추가하면 된다.

```
-- 가구 유형 추가
CREATE TABLE furniture (
    product_id BIGINT PRIMARY KEY,
    width INT NOT NULL,
    height INT NOT NULL,
    depth INT NOT NULL,
    assembly_required BOOLEAN DEFAULT FALSE,
    FOREIGN KEY (product_id) REFERENCES product(product_id)
);
```

단점

조인이 필요하다

상세 정보를 조회하려면 부모 테이블과 자식 테이블을 조인해야 한다. 조인은 매우 빠르지만, 너무 많은 데이터를 조인하면 성능에 영향을 줄 수 있다.

INSERT 시 두 번의 쿼리가 필요하다

데이터를 입력할 때 부모 테이블과 자식 테이블에 각각 INSERT해야 한다.

```
-- 1. 부모 테이블에 INSERT
INSERT INTO product (product_type, name, price, stock_quantity)
VALUES ('BOOK', '새로운 책', 25000, 50);

-- 2. 자식 테이블에 INSERT
INSERT INTO book (product_id, author, isbn, publisher)
VALUES (LAST_INSERT_ID(), '저자명', '1234567890123', '출판사');
```

쿼리가 복잡해질 수 있다

모든 상품의 상세 정보를 한 번에 조회하려면 여러 테이블을 LEFT JOIN해야 한다.

```
-- 모든 상품의 모든 정보 조회 (복잡한 쿼리)
SELECT
    p.*,
    b.author, b.isbn, b.publisher,
    e.manufacturer, e.warranty_months, e.power_consumption,
    c.size, c.color, c.material
FROM product p
LEFT JOIN book b ON p.product_id = b.product_id
LEFT JOIN electronics e ON p.product_id = e.product_id
LEFT JOIN clothing c ON p.product_id = c.product_id;
```

조인 전략 정리

장점	단점
정규화된 구조	조인이 필요함
NULL 값 없음	INSERT 시 두 번의 쿼리 필요
NOT NULL 제약조건 사용 가능	모든 상세 정보 조회 시 쿼리 복잡
외래 키 설정 가능	
유연한 확장 가능	

조인 전략을 선택하면 좋은 경우

- 상품 유형이 많거나 앞으로 계속 늘어날 예정일 때
- 각 유형별 고유 속성이 많을 때
- 데이터 무결성이 중요할 때
- 정규화된 구조를 원할 때

정리

학습 내용 정리

상속 관계 설계 - 문제 상황

- 객체지향 언어에는 상속 개념이 존재하지만, 관계형 데이터베이스(RDBMS)에는 상속 개념이 없다.
- 객체의 상속 구조(부모-자식)를 테이블로 표현하기 위해 **슈퍼타입-서브타입(Supertype/Subtype) 모델**을 사용한다.
- 쇼핑몰 상품 예시:
  - 부모(공통 속성): 상품명, 가격, 재고수량 등.
  - 자식(고유 속성): 도서(저자), 전자제품(제조사), 의류(사이즈) 등.
- 이를 해결하는 3가지 주요 전략(구현 클래스마다 테이블, 단일 테이블, 조인 전략)이 있다.

구현 클래스마다 테이블 전략

- 부모 클래스를 없애고, 자식 클래스마다 별도의 테이블을 생성하는 방식이다.
- 각 테이블(book, electronics, clothing)이 부모의 공통 속성(name, price 등)을 중복해서 모두 가진다.
- 가장 단순하고 무식한 방법이다.

구현 클래스마다 테이블 전략의 장단점

- **장점**
  - 구조가 단순하고 직관적이다.
  - 특정 유형만 조회할 때 성능이 좋다.
  - 각 테이블 별로 NOT NULL 제약조건을 걸 수 있다.
- **단점**
  - 전체 상품을 조회할 때 모든 테이블을 `UNION`로 묶어야 하므로 쿼리가 복잡하고 성능이 떨어진다.
  - 상품 ID만으로는 어떤 테이블을 봐야 할지 알 수 없어 모든 테이블을 찾아야 한다.
  - 다른 테이블(예: 주문)에서 외래 키(FK)를 설정할 수 없어 데이터 무결성을 보장하기 어렵다.
  - 공통 속성이 변경되면 모든 테이블을 수정해야 한다.
- **결론**: 치명적인 단점이 많아 실무에서는 거의 사용하지 않는다.

단일 테이블 전략

- 부모와 모든 자식의 속성을 하나의 테이블(`product`)에 통합하여 관리하는 방식이다.
- 상품 유형을 구분하기 위해 `product_type`(DTYPE) 컬럼을 필수로 사용한다.
- 자식 엔티티가 늘어나도 테이블은 하나만 존재한다.

단일 테이블 전략의 장단점

- **장점**
  - 조인이 필요 없어 조회 성능이 가장 우수하다.
  - 쿼리가 단순하고 직관적이다.
  - 외래 키 설정이 가능하며, 상품 ID가 전역적으로 유일하다.
- **단점**
  - 해당 유형이 아닌 컬럼은 모두 NULL로 저장되므로, NULL 값이 매우 많아진다.
  - 자식 속성에는 데이터베이스 레벨에서 NOT NULL 제약조건을 걸 수 없다.
  - 모든 속성을 저장하므로 테이블 크기가 비대해질 수 있다.
- **결론**: 구조가 단순하고 확장 가능성이 낮을 때 유용하다.

조인 전략

- 부모 테이블(공통 속성)과 자식 테이블(고유 속성)을 분리하고, 조인(JOIN)을 통해 데이터를 조회하는 방식이다.
- 가장 정규화된 정석적인 방법이다.
- **핵심 원리**: 자식 테이블의 기본 키(PK)가 부모 테이블의 기본 키를 참조하는 외래 키(FK) 역할도 겸한다. (1:1 관계)

조인 전략의 장단점

- **장점**

    - 데이터가 정규화되어 있어 중복이 없고 저장 공간이 효율적이다.
    - 불필요한 NULL 값이 발생하지 않는다.
    - 모든 컬럼에 대해 NOT NULL 제약조건을 활용할 수 있다.
    - 외래 키 참조가 용이하고, 새로운 유형 추가 시 확장성이 좋다.
- **단점**
    - 조회 시 조인이 필요하므로 성능에 영향을 줄 수 있다.
    - 데이터를 등록(INSERT)할 때 부모와 자식 테이블에 각각 쿼리를 날려야 하므로 2번 실행된다.
    - 모든 정보를 한 번에 조회하려면 쿼리가 복잡해진다(다중 LEFT JOIN).
- **결론**: 데이터 무결성이 중요하고 비즈니스가 복잡할 때 선택하는 가장 추천되는 방식이다.

상속 관계 전체 정리

상속 관계를 데이터베이스에서 표현하는 세 가지 전략을 모두 살펴보았다. 각 전략의 특징을 비교해보자.

전략별 비교표

특징	구현 클래스마다 테이블 전략	단일 테이블 전략	조인 전략
테이블 수	자식 수만큼	1개	1 + 자식 수
전체 조회	UNION ALL 필요	단순	공통 속성 단순
상세 조회	단순	단순	조인 필요
INSERT	1회	1회	2회
NULL 값	없음	많음	없음
NOT NULL 제약	가능	불가능	가능
외래 키	불가능	가능	가능
확장성	낮음	중간	높음
저장 효율	좋음	낮음	좋음

언제 어떤 전략을 선택할까?

구현 클래스마다 테이블 전략

- 일반적으로 권장하지 않는다

- 특수한 상황에서만 사용 (예: 레거시 시스템과의 호환)

단일 테이블 전략

- 상품 유형이 적고, 앞으로도 크게 늘어나지 않을 때
- 각 유형별 고유 속성이 적을 때
- 성능이 매우 중요하고 쿼리를 단순하게 유지하고 싶을 때
- NULL 값이 많아도 괜찮을 때

조인 전략

- 상품 유형이 많거나 계속 늘어날 예정일 때
- 각 유형별 고유 속성이 많을 때
- 데이터 무결성이 중요할 때
- 정규화된 깔끔한 구조를 원할 때

실무 가이드

실무에서는 데이터의 중요도와 조회 빈도, 확장성을 고려하여 **단일 테이블 전략**과 **조인 전략** 중 적절한 것을 선택해야 한다. 일반적으로 데이터의 구조가 단순하고 확장 가능성이 낮다면 단일 테이블 전략을, 비즈니스 로직이 복잡하고 데이터의 정합성이 중요하다면 조인 전략을 선택하는 것이 좋다. 조인 전략은 정규화된 구조로 데이터 무결성을 보장하고, 유연하게 확장할 수 있다. 조인으로 인한 성능 저하는 인덱스를 잘 설계하면 대부분 해결할 수 있다.

실무 선택 고려사항

단일 테이블 전략의 최고 장점은 단순하다는 것이다. 따라서 간단한 경우라면 단일 테이블 전략도 충분히 고려해야 한다.

- 유형 간 차이가 크고 고유 속성이 많으면: 조인 전략
- 유형 간 차이가 적고 고유 속성이 적으면: 단일 테이블 전략

ORM과 실무 개발 이야기

지금까지 배운 상속 관계 매핑 전략들을 보면서 "이걸 SQL로 일일이 다 짜야 하나?"라는 걱정이 들 수 있다. 특히 조인 전략의 경우 데이터를 넣을 때 INSERT를 두 번 해야 하고, 조회할 때 JOIN을 계속 걸어야 하니 코드를 작성하는 과정이 매우 번거롭게 느껴질 것이다.

현대 애플리케이션 개발에서는 이 문제를 **ORM(Object-Relational Mapping)** 기술로 해결한다. 자바 진영의 표준

기술인 **JPA(Java Persistence API)**가 대표적이다.

객체 지향 언어인 자바에서는 상속이 매우 자연스럽다. `Item`이라는 부모 클래스를 만들고 `Book`, `Album`, `Movie`가 이를 상속받게 코드를 작성한다. 그리고 JPA에게 "이 상속 관계를 **조인 전략**으로 매핑해줘"라고 설정 정보(어노테이션)만 넘기면 된다.
그러면 JPA가 애플리케이션 실행 시점에 우리가 배운 3가지 전략 중 하나에 맞춰 자동으로 테이블을 생성하거나 매핑한다. 개발자가 직접 복잡한 SQL을 작성하지 않아도, JPA가 데이터를 저장할 때 알아서 INSERT SQL을 두 번 나누어 실행하고, 조회할 때 필요한 JOIN 문을 생성해서 데이터를 가져온다. 만약 전략을 **단일 테이블 전략**으로 바꾸고 싶다면, 비즈니스 로직을 수정할 필요 없이 설정 옵션 하나만 변경하면 된다.

"그럼 DB 설계를 몰라도 되는 것 아닌가?"라고 생각할 수 있다. 절대 아니다. JPA는 개발을 편하게 도와주는 도구일 뿐이다. 결국 물리적인 테이블이 어떻게 생성되고, 어떤 쿼리가 실행되며, 그로 인해 성능에 어떤 영향을 미치는지 정확히 이해하고 있어야 올바른 전략을 선택할 수 있다. 우리가 오늘 이 내용을 깊이 있게 배운 이유가 바로 여기에 있다.

> **슈퍼타입-서브타입 모델(Supertype-Subtype Model)**
>
> 데이터 모델링 이론에서는 이러한 구조를 **슈퍼타입-서브타입 모델**이라고 부른다.
> - **슈퍼타입(Supertype)**: 모든 엔티티가 공통으로 가지는 속성을 모아둔 상위 엔티티다. 여기서는 `product` 엔티티가 해당된다.
> - **서브타입(Subtype)**: 개별 엔티티만의 고유한 속성을 가진 하위 엔티티다. `book`, `electronics`, `clothing` 엔티티가 해당된다.
>
> 논리적인 슈퍼타입 서브타입 모델을 지금까지 살펴본 3가지 방식으로 구현할 수 있다.
> 조인 전략은 이 논리적 모델을 물리적 테이블로 가장 정석적으로 구현한 방식이다. 여기서 핵심은 **자식 테이블(서브타입)의 기본 키(PK)가 부모 테이블(슈퍼타입)의 기본 키를 그대로 물려받아 사용한다**는 점이다.
> 즉, 자식 테이블의 `product_id`는 해당 행을 식별하는 **PK**인 동시에, 부모 테이블의 특정 행을 가리키는 **FK** 역할을 함께 수행한다. 이 때문에 두 테이블은 완벽한 **1:1 관계**가 성립하며, `ON p.product_id = b.product_id`와 같이 단순하게 PK끼리만 조인해도 데이터가 정확하게 연결된다.