9. 동적 설계 - EAV

#0.강의/2.데이터베이스로드맵/4.설계2

EAV 설계 - 기존 방식의 한계와 새로운 접근법

앞서 상속 관계 설계(슈퍼타입-서브타입 관계)에 대해서 학습했다. 상속 관계 설계는 강력한 도구이지만, 모든 상황에서 완벽한 해결책이 되지는 않는다. 이번 수업에서는 상속 관계로 해결되지 않는 문제 상황을 살펴보고, 이런 경우에 고려할 수 있는 EAV(Entity-Attribute-Value) 패턴에 대해 학습한다.

상속 관계 설계의 한계

상속 관계 설계는 엔티티 간의 계층 구조를 표현하는 좋은 방법이다. 하지만 다음과 같은 상황에서는 상속 관계만으로 해결하기 어려운 문제가 발생한다.

문제 상황 - 끊임없이 늘어나는 상품 속성

쇼핑몰에서 다양한 카테고리의 상품을 판매한다고 가정해보자. 처음에는 의류, 전자제품, 도서 3가지 카테고리만 있었다.

참고로 **각 카테고리는 서로 다른 고유의 속성**을 가진다고 가정하겠다.

```sql
-- 슈퍼타입: 상품 공통 정보
CREATE TABLE product (
    product_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(200) NOT NULL,
    price INT NOT NULL,
    category VARCHAR(50) NOT NULL,
    created_at DATETIME NOT NULL
);
```

```sql
-- 서브타입: 의류
CREATE TABLE product_clothing (
    product_id BIGINT PRIMARY KEY,
    size VARCHAR(10),
    color VARCHAR(50),
    material VARCHAR(100),
    FOREIGN KEY (product_id) REFERENCES product(product_id)
);

-- 서브타입: 전자제품
CREATE TABLE product_electronics (
    product_id BIGINT PRIMARY KEY,
    manufacturer VARCHAR(100),
    warranty_months INT,
    power_consumption VARCHAR(50),
    FOREIGN KEY (product_id) REFERENCES product(product_id)
);

-- 서브타입: 도서
CREATE TABLE product_book (
    product_id BIGINT PRIMARY KEY,
    author VARCHAR(100),
    publisher VARCHAR(100),
    isbn VARCHAR(20),
    page_count INT,
    FOREIGN KEY (product_id) REFERENCES product(product_id)
);
```

- 예시이다. 테이블을 생성하지는 말자.

이 구조는 처음에는 잘 작동한다. 하지만 시간이 지나면서 문제가 발생하기 시작한다.

첫 번째 문제 - 카테고리 폭발

사업이 성장하면서 카테고리가 점점 늘어난다.

- 가구, 식품, 스포츠용품, 화장품, 주방용품, 반려동물용품, 자동차용품...
- 6개월 후에는 50개의 카테고리가 생겼다.
- 1년 후에는 200개의 카테고리가 되었다.

카테고리가 늘어날 때마다 새로운 서브타입 테이블을 만들어야 한다. 200개의 카테고리라면 200개의 서브타입 테이블이 필요하다. 이것은 관리가 거의 불가능하다.

두 번째 문제 - 동적 속성 변경

카테고리별로 필요한 속성이 계속 변경된다.

- 의류에 "세탁방법", "원산지", "시즌" 속성이 추가되어야 한다.
- 전자제품에 "에너지효율등급", "KC인증번호"가 추가되어야 한다.
- 운영팀에서 매주 새로운 속성 추가를 요청한다.

속성이 추가될 때마다 `ALTER TABLE`로 컬럼을 추가해야 한다. 운영 중인 시스템에서 테이블 구조를 자주 변경하는 것은 위험하고 번거롭다.

세 번째 문제 - 예측 불가능한 속성

어떤 카테고리에는 완전히 예상치 못한 속성이 필요하다.

- 와인: 빈티지, 포도품종, 와이너리, 알코올도수, 음용온도, 디캔팅시간...
- 캠핑용품: 수용인원, 방수등급, 무게, 설치시간, 시즌적합성...
- 악기: 조율방식, 음역대, 재질, 제조국가, 연주난이도...

각 카테고리마다 전혀 다른 종류와 개수의 속성이 필요하다. 이런 속성들을 미리 예측해서 테이블 구조로 설계하는 것은 거의 불가능하다.

네 번째 문제 - 셀러마다 다른 속성

오픈마켓처럼 여러 셀러가 상품을 등록하는 경우를 생각해보자.

- 셀러 A는 휴대폰을 팔면서 "화면크기", "배터리용량", "카메라화소"를 입력하고 싶다.
- 셀러 B는 같은 휴대폰이지만 "5G지원여부", "듀얼심여부", "방수등급"을 입력하고 싶다.
- 셀러 C는 "색상옵션", "저장용량옵션", "번들구성"을 입력하고 싶다.

같은 카테고리라도 셀러마다 다른 속성을 입력하고 싶어한다. 모든 셀러의 요구사항을 테이블 컬럼으로 미리 정의해두는 것은 현실적으로 불가능하다.

상속 관계가 적합한 경우 vs 적합하지 않은 경우

상속 관계 설계와 앞으로 설명할 EAV 패턴 중 어떤 것을 선택해야 하는지 정리해보자.

상속 관계가 적합한 경우

- 서브타입의 종류가 명확하고 제한적이다 (예: 결제수단 - 카드/계좌이체/가상계좌)
- 각 서브타입의 속성이 안정적이고 자주 변경되지 않는다
- 서브타입별 속성에 대한 제약조건(NOT NULL, 타입검증 등)이 중요하다

- 서브타입별로 복잡한 조인이나 조회가 자주 필요하다

상속 관계가 적합하지 않은 경우

- 서브타입(카테고리)이 계속 늘어날 수 있다
- 속성이 동적으로 추가, 수정, 삭제되어야 한다
- 카테고리별로 필요한 속성을 미리 예측하기 어렵다
- 사용자(셀러, 관리자 등)가 직접 속성을 정의해야 한다
- 유연성이 데이터 무결성보다 더 중요하다

이렇게 상속 관계로는 해결하기 어려운 상황에서 EAV 패턴을 고려할 수 있다.
다음 시간에는 본격적으로 EAV 패턴에 대해서 알아보자.

EAV 패턴 소개

EAV는 Entity-Attribute-Value의 약자로, 데이터를 저장하는 특별한 방식이다. 일반적인 테이블 구조와는 완전히 다른 접근법을 사용한다.

전통적인 테이블 구조 vs EAV 구조

먼저 전통적인 방식과 EAV 방식의 차이를 이해해보자.

전통적인 테이블 구조

전통적인 방식에서는 각 속성이 테이블의 컬럼이 된다.

```
DROP TABLE IF EXISTS product_phone;

CREATE TABLE product_phone (
    product_id BIGINT PRIMARY KEY,
    name VARCHAR(200),
    screen_size VARCHAR(20),
    battery_capacity VARCHAR(20),
    camera_pixels VARCHAR(20),
    ram VARCHAR(20),
    storage VARCHAR(20)
```

```
);

INSERT INTO product_phone VALUES
(1, 'Galaxy S24', '6.2인치', '4000mAh', '50MP', '8GB', '256GB'),
(2, 'iPhone 15', '6.1인치', '3349mAh', '48MP', '6GB', '128GB');
```

[저장 결과]

product_id	name	screen_size	battery_capacity	camera_pixels	ram	s
1	Galaxy S24	6.2인치	4000mAh	50MP	8GB	2
2	iPhone 15	6.1인치	3349mAh	48MP	6GB	1

이 방식은 익숙하고 직관적이다. 하지만 새로운 속성(예: 5G지원여부)을 추가하려면 `ALTER TABLE`로 컬럼을 추가해야 한다.

EAV 구조

EAV 방식에서는 속성 이름과 값을 열(Column)이 아니라 행(row)으로 저장한다.

```
-- product와 연관된 기존 테이블
DROP TABLE IF EXISTS orders;
DROP TABLE IF EXISTS furniture;
DROP TABLE IF EXISTS book;
DROP TABLE IF EXISTS clothing;
DROP TABLE IF EXISTS electronics;

DROP TABLE IF EXISTS product_attribute;
DROP TABLE IF EXISTS product;

-- Entity: 상품
CREATE TABLE product (
    product_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(200) NOT NULL,
    created_at DATETIME NOT NULL
);

-- Attribute-Value: 상품의 속성과 값
```

```sql
CREATE TABLE product_attribute (
    attribute_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    product_id BIGINT NOT NULL,
    attr_name VARCHAR(100) NOT NULL,
    attr_value VARCHAR(500),
    FOREIGN KEY (product_id) REFERENCES product(product_id)
);

-- 데이터 입력
INSERT INTO product (product_id, name, created_at) VALUES
(1, 'Galaxy S24', NOW()),
(2, 'iPhone 15', NOW());

INSERT INTO product_attribute (product_id, attr_name, attr_value) VALUES
(1, 'screen_size', '6.2인치'),
(1, 'battery_capacity', '4000mAh'),
(1, 'camera_pixels', '50MP'),
(1, 'ram', '8GB'),
(1, 'storage', '256GB'),
(2, 'screen_size', '6.1인치'),
(2, 'battery_capacity', '3349mAh'),
(2, 'camera_pixels', '48MP'),
(2, 'ram', '6GB'),
(2, 'storage', '128GB');
```

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

[실행 결과]

product_id	name	created_at
1	Galaxy S24	2026-01-15 10:00:00
2	iPhone 15	2026-01-15 10:00:00

```sql
SELECT * FROM product_attribute ORDER BY product_id, attr_name;
```

[실행 결과]

attribute_id	product_id	attr_name	attr_value
2	1	battery_capacity	4000mAh
3	1	camera_pixels	50MP
4	1	ram	8GB
1	1	screen_size	6.2인치
5	1	storage	256GB
7	2	battery_capacity	3349mAh
8	2	camera_pixels	48MP
9	2	ram	6GB
6	2	screen_size	6.1인치
10	2	storage	128GB

속성이 컬럼이 아니라 행으로 저장되어 있다. 이것이 EAV 패턴의 핵심이다.

EAV의 이름 이해하기

EAV는 세 가지 요소로 구성된다.

- **Entity(엔티티)**: 속성을 가지는 주체 (예: 상품, 사용자, 주문)
- **Attribute(속성)**: 엔티티가 가지는 특성의 이름 (예: screen_size, battery_capacity)
- **Value(값)**: 해당 속성의 실제 값 (예: 6.2인치, 4000mAh)

전통적인 테이블에서는 Attribute가 컬럼명이 되고, Value가 셀의 값이 된다. EAV에서는 Attribute와 Value 모두 데이터로 저장된다. 컬럼과 값이 모두 데이터로 저장되는 것이다.

EAV의 핵심 특징

EAV 패턴의 가장 중요한 특징은 **스키마의 유연성**이다.

새로운 속성 추가가 자유롭다

5G 지원 여부 속성을 추가하고 싶다면? 그냥 새로운 행을 INSERT하면 된다.

```
INSERT INTO product_attribute (product_id, attr_name, attr_value) VALUES
(1, 'supports_5g', 'Y'),
(2, 'supports_5g', 'Y');
```

테이블 구조 변경 없이 새로운 속성이 추가되었다.

엔티티마다 다른 속성을 가질 수 있다

Galaxy S24에만 있는 속성을 추가할 수도 있다.

```
INSERT INTO product_attribute (product_id, attr_name, attr_value) VALUES
(1, 's_pen_support', 'N'),
(1, 'samsung_dex', 'Y');
```

iPhone에는 없는 속성이지만 문제없이 저장된다.

Galaxy S24의 속성 조회

```
SELECT p.name, pa.attr_name, pa.attr_value
FROM product p
JOIN product_attribute pa ON p.product_id = pa.product_id
WHERE p.product_id = 1
ORDER BY pa.attr_name;
```

[실행 결과]

name	attr_name	attr_value
Galaxy S24	battery_capacity	4000mAh
Galaxy S24	camera_pixels	50MP

Galaxy S24	ram	8GB
Galaxy S24	s_pen_support	N
Galaxy S24	samsung_dex	Y
Galaxy S24	screen_size	6.2인치
Galaxy S24	storage	256GB
Galaxy S24	supports_5g	Y

iPhone 15의 속성 조회

```
SELECT p.name, pa.attr_name, pa.attr_value
FROM product p
JOIN product_attribute pa ON p.product_id = pa.product_id
WHERE p.product_id = 2
ORDER BY pa.attr_name;
```

[실행 결과]

name	attr_name	attr_value
iPhone 15	battery_capacity	3349mAh
iPhone 15	camera_pixels	48MP
iPhone 15	ram	6GB
iPhone 15	screen_size	6.1인치
iPhone 15	storage	128GB
iPhone 15	supports_5g	Y

EAV 실습 - 쇼핑몰 상품 속성 관리

쇼핑몰에서 EAV 패턴을 어떻게 활용하는지 단계별로 실습해보자.

기본 테이블 구조 설계

먼저 실습을 위한 테이블을 생성한다.

```
DROP TABLE IF EXISTS product_attribute;
DROP TABLE IF EXISTS product;
DROP TABLE IF EXISTS category;

-- 카테고리 테이블
CREATE TABLE category (
    category_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    parent_id BIGINT,
    FOREIGN KEY (parent_id) REFERENCES category(category_id)
);

-- 상품 테이블 (Entity)
CREATE TABLE product (
    product_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    category_id BIGINT NOT NULL,
    name VARCHAR(200) NOT NULL,
    price INT NOT NULL,
    status VARCHAR(20) DEFAULT 'ACTIVE',
    created_at DATETIME NOT NULL,
    FOREIGN KEY (category_id) REFERENCES category(category_id)
);

-- 상품 속성 테이블 (Attribute-Value)
CREATE TABLE product_attribute (
    attribute_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    product_id BIGINT NOT NULL,
    attr_name VARCHAR(100) NOT NULL,
    attr_value VARCHAR(500),
    created_at DATETIME NOT NULL,
    FOREIGN KEY (product_id) REFERENCES product(product_id)
);
```

```sql
-- 인덱스 추가
CREATE INDEX idx_product_attribute_product ON product_attribute(product_id);
CREATE INDEX idx_product_attribute_name ON product_attribute(attr_name);
```

이 구조에서 `product` 테이블은 모든 상품이 공통으로 가지는 핵심 속성만 포함한다. 카테고리별로 다른 속성들은 `product_attribute` 테이블에 저장한다.

테스트 데이터 입력

다양한 카테고리의 상품을 입력해보자.

```sql
-- 카테고리 입력
INSERT INTO category (category_id, name, parent_id) VALUES
(1, '전자제품', NULL),
(2, '스마트폰', 1),
(3, '노트북', 1),
(4, '의류', NULL),
(5, '상의', 4),
(6, '식품', NULL);

-- 상품 입력
INSERT INTO product (product_id, category_id, name, price, created_at) VALUES
(1, 2, 'Galaxy S24 Ultra', 1650000, NOW()),
(2, 2, 'iPhone 15 Pro', 1550000, NOW()),
(3, 3, 'MacBook Pro 14', 2690000, NOW()),
(4, 5, '캐시미어 니트', 89000, NOW());

-- 스마트폰 속성 (Galaxy S24 Ultra)
INSERT INTO product_attribute (product_id, attr_name, attr_value, created_at)
VALUES
(1, 'screen_size', '6.8인치', NOW()),
(1, 'battery_capacity', '5000mAh', NOW()),
(1, 'ram', '12GB', NOW()),
(1, 'storage', '256GB', NOW()),
(1, 'main_camera', '200MP', NOW()),
(1, 'processor', 'Snapdragon 8 Gen 3', NOW()),
(1, 's_pen', 'Y', NOW()),
(1, 'color', '티타늄 바이올렛', NOW());
```

```sql
-- 스마트폰 속성 (iPhone 15 Pro)
INSERT INTO product_attribute (product_id, attr_name, attr_value, created_at)
VALUES
(2, 'screen_size', '6.1인치', NOW()),
(2, 'battery_capacity', '3274mAh', NOW()),
(2, 'ram', '8GB', NOW()),
(2, 'storage', '256GB', NOW()),
(2, 'main_camera', '48MP', NOW()),
(2, 'processor', 'A17 Pro', NOW()),
(2, 'action_button', 'Y', NOW()),
(2, 'color', '블루 티타늄', NOW());

-- 노트북 속성 (MacBook Pro 14)
INSERT INTO product_attribute (product_id, attr_name, attr_value, created_at)
VALUES
(3, 'screen_size', '14.2인치', NOW()),
(3, 'processor', 'M3 Pro', NOW()),
(3, 'ram', '18GB', NOW()),
(3, 'storage', '512GB SSD', NOW()),
(3, 'battery_life', '17시간', NOW()),
(3, 'weight', '1.61kg', NOW()),
(3, 'ports', 'HDMI, SD카드, MagSafe, Thunderbolt x3', NOW());

-- 의류 속성 (캐시미어 니트)
INSERT INTO product_attribute (product_id, attr_name, attr_value, created_at)
VALUES
(4, 'material', '캐시미어 100%', NOW()),
(4, 'size', 'S, M, L, XL', NOW()),
(4, 'color', '아이보리, 네이비, 그레이', NOW()),
(4, 'origin', '이탈리아', NOW()),
(4, 'washing', '드라이클리닝', NOW()),
(4, 'season', 'F/W', NOW());
```

상품 속성 조회하기

EAV 구조에서 데이터를 조회하는 다양한 방법을 알아보자.

특정 상품의 모든 속성 조회

```sql

SELECT
```

```
    p.product_id,
    p.name AS product_name,
    p.price,
    pa.attr_name,
    pa.attr_value
FROM product p
JOIN product_attribute pa ON p.product_id = pa.product_id
WHERE p.product_id = 1
ORDER BY pa.attr_name;
```

- `product`와 `product_attribute`를 조인해서 상품과 속성을 함께 구할 수 있다.

[실행 결과]

product_id	product_name	price	attr_name	attr_value
1	Galaxy S24 Ultra	1650000	battery_capacity	5000mAh
1	Galaxy S24 Ultra	1650000	color	티타늄 바이올렛
1	Galaxy S24 Ultra	1650000	main_camera	200MP
1	Galaxy S24 Ultra	1650000	processor	Snapdragon 8 Gen 3
1	Galaxy S24 Ultra	1650000	ram	12GB
1	Galaxy S24 Ultra	1650000	s_pen	Y
1	Galaxy S24 Ultra	1650000	screen_size	6.8인치
1	Galaxy S24 Ultra	1650000	storage	256GB

특정 속성으로 상품 검색

RAM이 12GB인 상품을 찾아보자.

```
SELECT
    p.product_id,
    p.name,
    p.price,
    pa.attr_value AS ram
FROM product p
JOIN product_attribute pa ON p.product_id = pa.product_id
```

```
WHERE pa.attr_name = 'ram'
  AND pa.attr_value = '12GB';
```

- `attr_name`, `attr_value` 두 조건을 함께 사용해야 한다. 만약 `attr_value`만 사용하면 `ram`이 아닌 다른 속성에 `12GB` 값도 결과에 포함될 수 있다.

[실행 결과]

product_id	name	price	ram
1	Galaxy S24 Ultra	1650000	12GB

여러 속성 조건으로 검색

화면 크기가 6인치 이상이면서 S펜을 지원하는 상품을 찾아보자. EAV 구조에서 여러 조건을 AND로 검색하려면 조금 복잡한 쿼리가 필요하다.

```
SELECT p.product_id, p.name, p.price
FROM product p
WHERE p.product_id IN (
    SELECT product_id
    FROM product_attribute
    WHERE attr_name = 'screen_size'
      AND attr_value LIKE '6%'
)
AND p.product_id IN (
    SELECT product_id
    FROM product_attribute
    WHERE attr_name = 's_pen'
      AND attr_value = 'Y'
);
```

[실행 결과]

product_id	name	price
1	Galaxy S24 Ultra	1650000

행을 열로 변환하기 (Pivot)

EAV 구조의 데이터를 전통적인 테이블 형태로 보고 싶을 때가 있다. 이럴 때는 피벗(Pivot) 기법을 사용한다.

스마트폰 상품을 일반 테이블 형태로 조회

```
SELECT
    p.product_id,
    p.name,
    p.price,
    MAX(CASE WHEN pa.attr_name = 'screen_size' THEN pa.attr_value END) AS
screen_size,
    MAX(CASE WHEN pa.attr_name = 'battery_capacity' THEN pa.attr_value END) AS
battery,
    MAX(CASE WHEN pa.attr_name = 'ram' THEN pa.attr_value END) AS ram,
    MAX(CASE WHEN pa.attr_name = 'storage' THEN pa.attr_value END) AS storage
FROM product p
LEFT JOIN product_attribute pa ON p.product_id = pa.product_id
WHERE p.category_id = 2  -- 스마트폰 카테고리
GROUP BY p.product_id, p.name, p.price;
```

[실행 결과]

product_id	name	price	screen_size	battery	ram	storage
1	Galaxy S24 Ultra	1650000	6.8인치	5000mAh	12GB	256GB
2	iPhone 15 Pro	1550000	6.1인치	3274mAh	8GB	256GB

이 쿼리는 행으로 저장된 속성들을 열로 변환하여 보여준다. `CASE`와 `MAX()`를 조합하여 Pivot을 구현한다.

> **피벗 테이블**
>
> 피벗 테이블에 대한 내용은 데이터베이스 기본편 - 6. CASE 문에서 학습했다. 해당 내용을 참고하자.

속성 동적 추가 실습

EAV의 가장 큰 장점인 동적 속성 추가를 실습해보자.

새로운 속성 추가 - 테이블 변경 없음

Galaxy S24 Ultra에 새로운 속성을 추가해보자.

```
-- 새로운 속성 추가
INSERT INTO product_attribute (product_id, attr_name, attr_value, created_at)
VALUES
(1, 'water_resistance', 'IP68', NOW()),
(1, 'wireless_charging', 'Y', NOW()),
(1, 'reverse_charging', 'Y', NOW()),
(1, '5g_support', 'Y', NOW());

-- 확인
SELECT attr_name, attr_value
FROM product_attribute
WHERE product_id = 1
ORDER BY created_at DESC
LIMIT 5;
```

[실행 결과]

attr_name	attr_value
5g_support	Y
reverse_charging	Y
wireless_charging	Y
water_resistance	IP68
...	...

`ALTER TABLE` 없이 새로운 속성이 추가되었다. 이것이 EAV의 핵심 장점이다.

EAV 패턴 개선 - 속성 정의 테이블

기본 EAV 구조는 유연하지만 몇 가지 문제가 있다. **속성명을 자유롭게 입력**할 수 있어서 오타가 발생할 수 있고, 어떤 속성들이 있는지 파악하기 어렵다. 이를 개선한 구조를 살펴보자.

속성 정의 테이블 추가

이 문제를 해결하기 위해 속성의 메타데이터를 관리하는 테이블을 추가한다.

```
DROP TABLE IF EXISTS product_attribute;
DROP TABLE IF EXISTS attribute_definition;

-- 속성 정의 테이블
CREATE TABLE attribute_definition (
    attr_def_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    category_id BIGINT,
    attr_name VARCHAR(100) NOT NULL,
    attr_label VARCHAR(100) NOT NULL,
    attr_type VARCHAR(20) NOT NULL,      -- TEXT, NUMBER, BOOLEAN, SELECT
    is_required BOOLEAN DEFAULT FALSE,
    is_searchable BOOLEAN DEFAULT FALSE,
    display_order INT DEFAULT 0,
    options VARCHAR(500),                -- SELECT 타입일 때 선택 옵션
    created_at DATETIME NOT NULL,
    FOREIGN KEY (category_id) REFERENCES category(category_id)
);

-- 상품 속성 테이블 (개선)
CREATE TABLE product_attribute (
    attribute_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    product_id BIGINT NOT NULL,
    attr_def_id BIGINT NOT NULL,
    attr_value VARCHAR(500),
    created_at DATETIME NOT NULL,
    FOREIGN KEY (product_id) REFERENCES product(product_id),
    FOREIGN KEY (attr_def_id) REFERENCES attribute_definition(attr_def_id)
);

CREATE INDEX idx_product_attribute_product ON product_attribute(product_id);
```

```
CREATE INDEX idx_product_attribute_def ON product_attribute(attr_def_id);
```

구조의 변화

- `attr_name`을 직접 저장하는 대신 `attr_def_id`로 속성 정의를 참조한다
- 속성의 타입, 필수 여부, 검색 가능 여부 등 메타데이터를 관리할 수 있다
- `category_id`를 참조해서 카테고리별로 어떤 속성들이 있는지 명확하게 정의된다
- `product_attribute` 테이블은 이제 속성의 이름 대신에 `attr_def_id`를 통해서 미리 정의된 속성을 참조한다.
- **참고**: 각 상품의 속성의 중복 입력을 막으려면 `product_attribute` 테이블의 `product_id`, `attr_def_id` 컬럼을 묶어서 유니크 제약조건을 적용하면 된다.

속성 정의 데이터 입력

```
-- 스마트폰 카테고리 속성 정의
INSERT INTO attribute_definition
(category_id, attr_name, attr_label, attr_type, is_required, is_searchable,
display_order, options, created_at) VALUES
(2, 'screen_size', '화면 크기', 'TEXT', TRUE, TRUE, 1, NULL, NOW()),
(2, 'battery_capacity', '배터리 용량', 'TEXT', TRUE, TRUE, 2, NULL, NOW()),
(2, 'ram', 'RAM', 'SELECT', TRUE, TRUE, 3, '4GB,6GB,8GB,12GB,16GB', NOW()),
(2, 'storage', '저장 용량', 'SELECT', TRUE, TRUE, 4,
'64GB,128GB,256GB,512GB,1TB', NOW()),
(2, 'processor', '프로세서', 'TEXT', TRUE, FALSE, 5, NULL, NOW()),
(2, 'main_camera', '메인 카메라', 'TEXT', FALSE, TRUE, 6, NULL, NOW()),
(2, 'color', '색상', 'TEXT', TRUE, TRUE, 7, NULL, NOW()),
(2, 'water_resistance', '방수 등급', 'SELECT', FALSE, TRUE, 8, 'IP67,IP68',
NOW()),
(2, '5g_support', '5G 지원', 'BOOLEAN', FALSE, TRUE, 9, NULL, NOW());

-- 의류 카테고리 속성 정의
INSERT INTO attribute_definition
(category_id, attr_name, attr_label, attr_type, is_required, is_searchable,
display_order, options, created_at) VALUES
(5, 'material', '소재', 'TEXT', TRUE, TRUE, 1, NULL, NOW()),
(5, 'size', '사이즈', 'SELECT', TRUE, TRUE, 2, 'XS,S,M,L,XL,XXL', NOW()),
(5, 'color', '색상', 'TEXT', TRUE, TRUE, 3, NULL, NOW()),
(5, 'origin', '원산지', 'TEXT', FALSE, FALSE, 4, NULL, NOW()),
(5, 'washing', '세탁 방법', 'TEXT', FALSE, FALSE, 5, NULL, NOW()),
(5, 'season', '시즌', 'SELECT', FALSE, TRUE, 6, 'S/S,F/W,사계절', NOW());
```

개선된 구조로 데이터 조회

카테고리별 속성 정의 조회

스마트폰의 카테고리별 속성 정의를 확인해보자.

```
SELECT
    ad.attr_name,
    ad.attr_label,
    ad.attr_type,
    CASE WHEN ad.is_required THEN 'Y' ELSE 'N' END AS required,
    CASE WHEN ad.is_searchable THEN 'Y' ELSE 'N' END AS searchable,
    ad.options
FROM attribute_definition ad
JOIN category c ON ad.category_id = c.category_id
WHERE c.name = '스마트폰'
ORDER BY ad.display_order;
```

[실행 결과]

attr_name	attr_label	attr_type	required	searchable	options
screen_size	화면 크기	TEXT	Y	Y	NULL
battery_capacity	배터리 용량	TEXT	Y	Y	NULL
ram	RAM	SELECT	Y	Y	4GB,6GB,8GB,12GB,16GB
storage	저장 용량	SELECT	Y	Y	64GB,128GB,256GB,512GB,1TB
processor	프로세서	TEXT	Y	N	NULL
main_camera	메인 카메라	TEXT	N	Y	NULL

color	색상	TEXT	Y	Y	NULL
water_resistance	방수 등급	SELECT	N	Y	IP67,IP68
5g_support	5G 지원	BOOLEAN	N	Y	NULL

- 참고: `options`를 자주 사용한다면 정규화해서 별도의 테이블로 분리하는 것이 좋다. 여기서는 예시를 단순하게 설명하기 위해 CSV 스타일로 저장해두었다.

`attribute_definition`에 미리 속성을 정의해둔 덕분에 이제 스마트폰 카테고리에 어떤 속성들이 있고, 각 속성의 특성이 무엇인지 명확하게 알 수 있다.

상품 등록 화면 데이터 제공

속성 정의 테이블을 도입하면 추가적인 이점도 얻을 수 있는데 대표적으로 다음과 같은 이점이 있다.

관리자가 상품을 등록할 때, 해당 카테고리에 필요한 속성 목록을 제공해야 한다.

```
-- 스마트폰 카테고리 상품 등록 시 필요한 속성 폼 정보
SELECT
    ad.attr_def_id,
    ad.attr_name,
    ad.attr_label,
    ad.attr_type,
    ad.is_required,
    ad.options
FROM attribute_definition ad
WHERE ad.category_id = 2   -- 스마트폰
ORDER BY ad.display_order;
```

[실행 결과]

attr_def_id	attr_name	attr_label	attr_type	is_required	options
1	screen_size	화면 크기	TEXT	1	NULL

2	battery_capacity	배터리 용량	TEXT	1	NULL
3	ram	RAM	SELECT	1	4GB,6GB,8GB,12GB,16GB
4	storage	저장 용량	SELECT	1	64GB,128GB,256GB,512GB,1TB
5	processor	프로세서	TEXT	1	NULL
6	main_camera	메인 카메라	TEXT	0	NULL
7	color	색상	TEXT	1	NULL
8	water_resistance	방수 등급	SELECT	0	IP67,IP68
9	5g_support	5G 지원	BOOLEAN	0	NULL

이 정보를 바탕으로 프론트엔드에서 동적으로 입력 폼을 생성할 수 있다. `attr_type`에 따라 텍스트 입력, 셀렉트 박스, 체크박스 등을 렌더링한다.
물론 속성 정의 테이블에서 사용한 다양한 컬럼들은 비즈니스 요구사항마다 달라진다. 여러분의 환경에 맞게 설계하면 된다.

개선된 구조에 속성 값 입력

개선된 구조에 데이터를 입력해보자. 이제 `attr_name` 대신 `attr_def_id` 사용한다.

```
INSERT INTO product_attribute (product_id, attr_def_id, attr_value,
created_at) VALUES
(1, 1, '6.8인치', NOW()),     -- screen_size (id:1)
(1, 2, '5000mAh', NOW()),     -- battery_capacity (id:2)
(1, 3, '12GB', NOW());        -- ram (id:3)
```

저장된 데이터

```
SELECT * FROM product_attribute;
```

[실행 결과]

attribute_id	product_id	attr_def_id	attr_value	created_at
1	1	1	6.8인치	...
2	1	2	5000mAh	...
3	1	3	12GB	...

- 이제 `attr_name` 대신 `attr_def_id` 사용하는 것을 확인할 수 있다.
- `attr_name`의 중복이 제거되고, 정규화 된다.

개선된 구조의 데이터 조회

이제 개선된 구조에서 데이터를 조회해보자. `product_attribute` 테이블에는 속성의 구체적인 이름이 없고 `attr_def_id`만 존재하므로, `attribute_definition` 테이블까지 조인(JOIN)해야 사람이 이해할 수 있는 정보를 볼 수 있다.

```
SELECT
    p.name AS product_name,
    ad.attr_label,
    pa.attr_value
FROM product p
JOIN product_attribute pa ON p.product_id = pa.product_id
JOIN attribute_definition ad ON pa.attr_def_id = ad.attr_def_id
WHERE p.product_id = 1
ORDER BY ad.display_order;
```

[실행 결과]

product_name	attr_label	attr_value
Galaxy S24 Ultra	화면 크기	6.8인치
Galaxy S24 Ultra	배터리 용량	5000mAh
Galaxy S24 Ultra	RAM	12GB

조회 쿼리에서 조인이 하나 늘어났다. 하지만 `attr_name`을 직접 입력하던 이전 방식과 비교했을 때, `attr_label`(화면 크기)처럼 사용자에게 보여주기 위한 이름을 별도로 관리할 수 있고, 오타로 인한 데이터 파편화를 막을 수 있다는 강력한 장점이 생겼다.

실무에서는 이렇게 **메타데이터(속성 정의)와 실제 데이터(속성 값)를 분리**하여 시스템의 유연성과 데이터 무결성을 동시에 잡는 설계를 자주 사용한다.

정리
지금까지 쇼핑몰의 다양한 상품 속성을 처리하기 위해 EAV 패턴을 배우고, 이를 메타데이터 테이블을 통해 개선하는 과정까지 실습했다.

- **EAV 패턴**: 테이블 구조 변경 없이 속성을 동적으로 추가할 수 있다.
- **메타데이터 관리**: 속성 정의 테이블을 두어 데이터의 타입, 필수 여부, 화면 표시 순서 등을 체계적으로 관리한다.

이 구조를 잘 이해하고 있으면, 변화가 잦은 비즈니스 환경에서도 유연하게 대처할 수 있는 데이터베이스를 설계할 수 있다.

EAV의 장단점과 사용 시 주의사항

EAV 패턴은 강력한 유연성을 제공하지만, 그에 따른 대가가 크다. 따라서 장단점을 명확히 이해하고 적절한 상황에서 사용해야 한다.

EAV의 장점

스키마 유연성
가장 큰 장점은 테이블 구조 변경 없이 새로운 속성을 추가할 수 있다는 것이다. 운영 중인 시스템에서 `ALTER TABLE`을 실행하는 것은 위험하고 시간이 오래 걸릴 수 있다. EAV는 이 문제를 완전히 해결한다.

희소 데이터 효율성

전통적인 테이블에서 100개의 속성 컬럼이 있는데, 각 상품이 평균 10개의 속성만 사용한다면? 90개의 컬럼에는 NULL이 저장되어 공간이 낭비된다. EAV는 실제 값이 있는 속성만 저장하므로 희소 데이터에 효율적이다.

엔티티별 다른 속성 지원

같은 카테고리라도 상품마다 다른 속성을 가질 수 있다. 셀러가 원하는 속성을 자유롭게 추가할 수 있다.

런타임 속성 정의

관리자 화면에서 새로운 속성을 정의하고 바로 사용할 수 있다. 개발자의 도움 없이 운영팀이 직접 속성을 관리할 수 있다.

EAV의 단점

쿼리 복잡성

여러 속성을 조회하거나 조건으로 검색하려면 복잡한 조인이나 서브쿼리가 필요하다.

```
-- 전통적 테이블: 간단한 쿼리
SELECT * FROM product_phone
WHERE screen_size >= '6.0' AND ram = '12GB';

-- EAV: 복잡한 쿼리
SELECT p.* FROM product p
WHERE p.product_id IN (
    SELECT product_id FROM product_attribute
    WHERE attr_name = 'screen_size' AND attr_value >= '6.0'
)
AND p.product_id IN (
    SELECT product_id FROM product_attribute
    WHERE attr_name = 'ram' AND attr_value = '12GB'
);
```

데이터 타입 제약 부족

모든 값이 VARCHAR로 저장되므로 데이터베이스 수준의 타입 검증이 불가능하다. 숫자여야 하는 값에 문자열이 들어갈 수 있다.

```
-- 잘못된 데이터도 입력 가능
INSERT INTO product_attribute (product_id, attr_name, attr_value)
VALUES (1, 'price_discount', '할인중');  -- 숫자여야 하는데 문자열
```

애플리케이션 레벨에서 검증 로직을 구현해야 한다.

참조 무결성 제약 어려움

외래 키 제약조건을 걸기 어렵다. 예를 들어, 색상 값이 특정 색상 테이블을 참조해야 한다면 EAV에서는 이를 데이터베이스 수준에서 강제할 수 없다.

인덱싱 제한

특정 속성에 대한 효율적인 인덱스 생성이 어렵다. 전통적인 테이블에서는 `CREATE INDEX idx_ram ON product_phone(ram)`으로 RAM 컬럼에 인덱스를 생성할 수 있다. EAV에서는 모든 속성이 같은 컬럼에 저장되므로 특정 속성만을 위한 인덱스를 만들기 어렵다.

조인 성능 저하

속성이 많아질수록 쿼리나 다중 조인의 성능이 떨어진다.

실무 팁: 데이터 타입별 컬럼 분리

EAV 패턴의 가장 큰 약점인 데이터 타입 정합성 문제를 해결하기 위해, 실무에서는 값(Value)을 저장하는 컬럼을 데이터 타입별로 분리하기도 한다. 모든 값을 하나의 `VARCHAR` 컬럼에 억지로 넣는 대신, 문자형, 숫자형, 날짜형 등으로 컬럼을 나누어 설계하는 방식이다.

타입별 컬럼을 분리한 EAV 테이블 설계

```
DROP TABLE IF EXISTS product_attribute_typed;

CREATE TABLE product_attribute_typed (
    attribute_id    BIGINT          NOT NULL AUTO_INCREMENT,
    product_id      BIGINT          NOT NULL,
    attr_name       VARCHAR(100)    NOT NULL,
    val_text        VARCHAR(255)    NULL, -- 문자열 값 저장
    val_int         INT             NULL, -- 숫자 값 저장
    val_date        DATETIME        NULL, -- 날짜 값 저장
    PRIMARY KEY (attribute_id)
);
```

- 이 예제는 단순하게 설명하기 위해 `attr_def_id` 대신에 `attr_name`을 그대로 사용했다.

데이터 저장 예시

```
-- 화면 패널 종류 (문자열)
INSERT INTO product_attribute_typed (product_id, attr_name, val_text)
VALUES (1, 'panel_type', 'OLED');

-- 배터리 용량 (숫자)
INSERT INTO product_attribute_typed (product_id, attr_name, val_int)
VALUES (1, 'battery_capacity', 5000);
```

[실행 결과]

attribute_id	product_id	attr_name	val_text	val_int	val_date
1	1	panel_type	OLED	NULL	NULL
2	1	battery_capacity	NULL	5000	NULL

이렇게 컬럼을 분리하면 다음과 같은 이점이 있다.

1. **데이터 타입 보장**: 숫자 컬럼에 문자가 들어가는 것을 DB 차원에서 막을 수 있다.
2. **연산 및 정렬 효율성**: 숫자는 숫자로, 날짜는 날짜로 저장되므로 `ORDER BY` 정렬이나 `BETWEEN` 같은 범위 검색 시 올바르게 작동하며 인덱스도 효율적으로 탈 수 있다. 예를 들어, 문자열로 저장된 '10'과 '2'를 비교하면 '10'이 더 작다고 판단될 수 있지만, 숫자형 컬럼을 따로 쓰면 이런 문제를 피할 수 있다.

물론, 하나의 로우(Row)에 사용하지 않는 나머지 타입 컬럼들은 `NULL`로 채워지게 된다는 점은 감수해야 한다. 하지만 데이터 무결성이 중요한 커머스 환경에서는 매우 유용한 타협안이 될 수 있다.

EAV 사용 시 주의사항

핵심 속성은 EAV에 넣지 말 것

자주 조회되고, 검색 조건으로 사용되며, 데이터 무결성이 중요한 속성은 메인 테이블의 컬럼으로 정의한다.

```sql
-- 좋은 예: 핵심 속성은 product 테이블에
CREATE TABLE product (
    product_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    category_id BIGINT NOT NULL,
    name VARCHAR(200) NOT NULL,
    price INT NOT NULL,           -- 핵심 속성
    brand VARCHAR(100),           -- 핵심 속성
    status VARCHAR(20),           -- 핵심 속성
    created_at DATETIME NOT NULL,
    FOREIGN KEY (category_id) REFERENCES category(category_id)
);

-- 부가 속성만 EAV로
CREATE TABLE product_attribute (
    attribute_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    product_id BIGINT NOT NULL,
    attr_def_id BIGINT NOT NULL,
    attr_value VARCHAR(500),
    FOREIGN KEY (product_id) REFERENCES product(product_id)
);
```

- 핵심 속성은 반드시 메인 테이블의 컬럼으로 사용해야 한다.
- 부가 속성만 EAV를 사용한다.

하이브리드 접근

완전한 EAV보다는 핵심 속성 + EAV의 하이브리드 방식을 권장한다. 자주 사용되는 속성은 컬럼으로, 동적으로 변하는 속성은 EAV로 관리한다.

EAV 실무 활용 사례

실무에서 EAV 패턴이 어떻게 활용되는지 몇 가지 사례를 살펴보자.

쇼핑몰 상품 상세 스펙

앞서 살펴본 것처럼, 오픈마켓이나 쇼핑몰에서 다양한 카테고리의 상품 스펙을 관리할 때 EAV를 사용한다.

설문조사/폼 빌더

동적으로 질문을 생성하고 응답을 저장하는 시스템에서 EAV가 유용하다.

```
-- 설문 정의
CREATE TABLE survey (
    survey_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    title VARCHAR(200) NOT NULL,
    created_at DATETIME NOT NULL
);

-- 질문 정의 (Attribute Definition)
CREATE TABLE survey_question (
    question_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    survey_id BIGINT NOT NULL,
    question_text VARCHAR(500) NOT NULL,
    question_type VARCHAR(20) NOT NULL,  -- TEXT, RADIO, CHECKBOX
    options VARCHAR(500),
    display_order INT,
    FOREIGN KEY (survey_id) REFERENCES survey(survey_id)
);

-- 응답 저장 (EAV)
CREATE TABLE survey_response (
    response_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    survey_id BIGINT NOT NULL,
    respondent_id BIGINT,
    question_id BIGINT NOT NULL,
    answer_value VARCHAR(1000),
    created_at DATETIME NOT NULL,
    FOREIGN KEY (survey_id) REFERENCES survey(survey_id),
    FOREIGN KEY (question_id) REFERENCES survey_question(question_id)
);
```

의료 시스템의 환자 데이터

환자마다 다른 검사 항목, 처방 내역, 증상 기록 등을 저장할 때 EAV가 유용하다. 의료 분야에서는 검사 항목이 수천 가

지가 될 수 있고, 환자마다 받는 검사가 다르기 때문이다.

사용자 프로필 확장 속성

기본 사용자 정보 외에 각 사용자 마다 다른 추가 프로필 정보를 유연하게 관리할 때 사용한다.

```
-- 기본 사용자 정보
CREATE TABLE users (
    user_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    email VARCHAR(100) NOT NULL UNIQUE,
    name VARCHAR(50) NOT NULL,
    created_at DATETIME NOT NULL
);

-- 확장 프로필 (EAV)
CREATE TABLE user_profile_attribute (
    attr_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    user_id BIGINT NOT NULL,
    attr_name VARCHAR(100) NOT NULL,
    attr_value VARCHAR(500),
    FOREIGN KEY (user_id) REFERENCES users(user_id)
);

-- 사용 예
INSERT INTO user_profile_attribute (user_id, attr_name, attr_value) VALUES
(1, 'github_url', 'https://github.com/user1'),
(1, 'blog_url', 'https://blog.user1.com'),
(1, 'preferred_language', 'ko'),
(2, 'company', 'ABC Corp'),
(2, 'job_title', 'Senior Developer');
```

정리

학습 내용 정리

EAV 설계 - 기존 방식의 한계와 새로운 접근법

- **상속 관계 설계의 한계**: 슈퍼타입-서브타입 구조는 카테고리가 폭발적으로 늘어나거나 속성이 동적으로 계속 변경되는 환경에서는 관리하기 어렵다.
- **주요 문제 상황**:
  - **카테고리 폭발**: 수백 개의 카테고리마다 테이블을 만드는 것은 불가능하다.
  - **동적 속성 변경**: 속성이 추가될 때마다 `ALTER TABLE`을 수행하는 것은 운영상 위험하다.
  - **예측 불가능성**: 와인, 캠핑용품 등 카테고리마다 전혀 다른 속성이 필요하다.
  - **사용자 정의 속성**: 셀러마다 다른 속성을 입력하고 싶어 하는 요구사항을 수용하기 어렵다.
- **설계 선택 기준**:
  - 속성이 안정적이고 데이터 무결성이 중요하면 **상속 관계**를 사용한다.
  - 속성이 동적이고 유연성이 중요하면 **EAV 패턴**을 고려한다.

EAV 패턴 소개

- **EAV의 개념**: Entity(엔티티), Attribute(속성), Value(값)의 약자로, 속성을 열(Column)이 아닌 행(Row)으로 저장하는 방식이다.
- **구조의 차이**:
  - 전통적 방식: 속성이 테이블의 컬럼이 된다. (스키마 변경 필요)
  - EAV 방식: 속성 이름과 값을 데이터(행)로 저장한다. (스키마 변경 불필요)
- **핵심 특징**:
  - **스키마 유연성**: `INSERT` 만으로 새로운 속성을 즉시 추가할 수 있다.
  - **엔티티별 차별화**: 같은 테이블(엔티티)에 속한 데이터라도 서로 다른 속성 집합을 가질 수 있다.

EAV 실습 - 쇼핑몰 상품 속성 관리

- **테이블 설계**:
  - `product`: 모든 상품의 공통 정보(이름, 가격 등)를 저장한다.
  - `product_attribute`: 상품별로 다른 속성(화면 크기, 소재 등)을 행으로 저장한다.
- **데이터 조회**:
  - 특정 상품의 속성을 조회하려면 `product`와 `product_attribute`를 조인한다.
  - 여러 속성 조건을 동시에 만족하는 상품을 찾으려면 복잡한 서브쿼리나 다중 조인이 필요하다.
- **피벗(Pivot)**: 행으로 저장된 EAV 데이터를 보기 편하게 열 형태로 변환할 때 `CASE WHEN`과 `MAX()` 등을 사용한다.
- **동적 추가**: `ALTER TABLE` 없이 데이터 입력만으로 5G 지원 여부, 방수 등급 등의 새로운 속성을 유연하게 추가할 수 있다.

EAV 패턴 개선 - 속성 정의 테이블

- **기본 EAV의 문제점**: 속성명을 문자열로 직접 입력하면 오타가 발생하기 쉽고, 어떤 속성이 존재하는지 관리하기 어렵다.

- **속성 정의 테이블(`attribute_definition`) 도입**:
  - 속성의 메타데이터(이름, 라벨, 타입, 필수 여부, 옵션 등)를 별도 테이블로 관리한다.
  - 속성 값 저장 시 속성 이름 대신 정의된 ID를 참조하여 정합성을 높인다.
- **장점**:
  - 카테고리별로 필요한 속성 목록을 명확히 정의할 수 있다.
  - 메타데이터를 활용해 프론트엔드에서 입력 폼(텍스트, 셀렉트 박스 등)을 동적으로 생성할 수 있다.

EAV의 장단점과 사용 시 주의사항

- **장점**:
  - 스키마 변경 없이 속성 확장이 자유롭다.
  - 실제 값이 있는 속성만 저장하므로 희소 데이터 처리에 효율적이다.
  - 런타임에 관리자가 직접 속성을 정의하고 운영할 수 있다.
- **단점**:
  - 쿼리가 복잡해지고 조인으로 인해 성능이 저하될 수 있다.
  - 모든 값이 문자열로 저장되어 DB 차원의 타입 검증이 불가능하다.
  - 인덱싱이 제한적이며 외래 키 제약 조건을 걸기 어렵다.
- **실무 팁**:
  - **타입별 컬럼 분리**: 값 저장 컬럼을 `val_text`, `val_int`, `val_date` 등으로 나누어 데이터 타입 정합성과 정렬 효율을 확보한다.
  - **하이브리드 접근**: 검색이 잦고 중요한 핵심 속성은 메인 테이블 컬럼으로, 가변적인 부가 속성만 EAV로 관리하는 것이 좋다.

EAV 실무 활용 사례

- **쇼핑몰 상품 스펙**: 카테고리마다 제각각인 상품 상세 정보를 저장할 때 사용한다.
- **설문조사 시스템**: 질문과 답변 형식이 동적으로 생성되는 환경에 적합하다.
- **의료 데이터**: 환자마다 다른 수많은 검사 항목과 결과를 저장할 때 유용하다.
- **사용자 프로필 확장**: 기본 정보 외에 사용자가 임의로 추가하는 프로필 정보를 관리할 때 사용한다.

EAV 선택에 대한 조언- 최후의 수단 (중요)

마지막으로, 실무자로서 EAV 패턴 선택에 대해 아주 중요한 조언을 하겠다.

"가급적이면 EAV를 사용하지 마라."

이 말이 모순처럼 들릴 수 있겠지만, 이것이 정답이다. EAV는 관계형 데이터베이스(RDBMS)가 제공하는 강력한 이점들을 대부분 포기하는 방식이다.

1. **데이터 무결성 포기**: 데이터베이스 레벨에서 `NOT NULL`, `INT`, `DATE` 같은 타입 체크나 외래 키 제약조건을 걸 수 없다. 잘못된 데이터가 들어가는 것을 막기 위해 애플리케이션 코드가 매우 복잡해진다.
2. **SQL 복잡도 증가**: 단순한 조회조차 수많은 조인(Join)과 서브쿼리가 필요하다. 쿼리를 작성하기도 어렵고, 유지보수하기도 힘들다.
3. **성능 저하**: 대용량 데이터에서 특정 속성으로 검색하거나 정렬할 때 인덱스를 효율적으로 태우기 어렵다.

그럼 언제 사용해야 하는가?

기존의 전통적인 테이블 설계(컬럼 추가, 정규화, 상속 관계 등)로는 도저히 해결할 수 없는 **"불확실성이 확실한"** 상황에서만 사용해야 한다. **단순히 "나중에 속성이 바뀔지도 몰라"라는 막연한 걱정으로 EAV를 도입해서는 안 된다.**

현대적인 대안: JSON Type

아마도 JSON 형식을 알고 있다면, EAV 대신에 JSON 형식으로 저장하면 더 효율적일 것 같다는 생각이 들 것이다. 만약 어쩔 수 없이 EAV 방식을 고려해야 한다면, 그리고 최신 데이터베이스를 사용한다면 JSON 방식을 고려해보자. 최신 RDBMS는 JSON 데이터에 대한 인덱싱과 검색 기능을 제공하므로, EAV의 복잡성을 줄이면서 유연성을 확보할 수 있다.

따라서 EAV 대신 **JSON 컬럼**을 사용하는 것이 더 효율적이다.

```
CREATE TABLE product (
    product_id BIGINT PRIMARY KEY,
    name VARCHAR(200),
    attributes JSON  -- 속성을 JSON으로 저장
);
```

데이터베이스 설계의 기본은 언제나 명확한 테이블과 컬럼이다. 기본에 충실하되, 정말 어쩔 수 없는 상황에서만 EAV라는 무기를 고려하자.

다음시간에는 JSON 설계에 대해서 알아보겠다.