10. 정규화

#0.강의/2.데이터베이스로드맵/3.설계1

정규화 - 시작

데이터베이스 설계에서 '정규화(Normalization)'는 가장 중요한 개념 중 하나다. 정규화는 데이터의 중복을 최소화하고, 데이터 일관성을 보장하며, 데이터 모델을 더 유연하게 만들기 위한 과정이다. 한마디로, **잘 설계된 데이터베이스를 만들기 위한 체계적인 절차**라고 할 수 있다.

우리는 앞에서 이미 잘 설계된 `member`, `product`, `orders`, `order_item` 테이블을 만들었다. 하지만 왜 테이블을 그렇게 여러 개로 나누어야만 했을까? 모든 데이터를 그냥 하나의 큰 테이블에 저장하면 더 편하지 않을까?

이 질문에 답하기 위해, 일부러 잘못 설계된 테이블에서부터 시작해서 점차적으로 개선해 나가는 과정을 밟아볼 것이다. 이 과정을 통해 정규화의 필요성을 몸소 체감하게 될 것이다.

> **정규화(Normalization) 용어**
>
> **정규화(Normalization)**라는 용어는 관계형 데이터베이스 모델의 창시자인 **에드거 F. 커드(Edgar F. Codd)**가 처음 제안했다. 이 개념은 수학과 논리학에서 사용되는 **정규형(Normal Form)**에서 영감을 받은 것이다.
>
> 수학에서 정규형이란 어떤 대상을 **가장 단순하고 표준적인 형태로 변환한 것**을 의미한다.
> 예를 들어, 분수 **2/4와 1/2**은 같은 값을 나타내지만, **1/2**이 가장 단순하고 명확한 형태이므로 이를 대표로 둔다. 이것이 바로 정규형이다.
>
> 데이터베이스에서의 정규화도 비슷한 맥락이다.
>
> - **정규화(Normalization)**란 데이터를 **정규형(Normal Form)**이라는 규칙에 맞추어 변환하는 과정이다.

- 이렇게 하면 데이터가 **중복되지 않고**, **일관성과 무결성을 유지**하며, **이상 현상(Anomaly)**을 방지할 수 있다.
- 따라서 "정규"라는 말은 **정규형의 조건을 만족하는 이상적 상태**를 의미한다.

문제 상황 제시: 하나의 거대한 테이블

만약 우리 쇼핑몰의 모든 주문 관련 데이터를 아래와 같은 하나의 테이블에 전부 저장한다고 상상해 보자.

`disaster_orders` **(재앙적인 주문 테이블)**

order_id	ordered_at	member_id	member_name	member_address	product_info
1001	2025-08-20	1	션	서울시 송파구	10:노트북:2:1500000, 15:키보드:1:50000
1002	2025-08-21	2	네이트	경기도 성남시	10:노트북:1:1500000
1003	2025-08-21	1	션	서울시 송파구	20:마우스:1:30000

우리는 이미 앞서 설계의 중요성을 통해 이런 설계가 얼마나 많은 재앙을 가져오는지 배웠다. 그래서 여기서는 간단히 설명하겠다.

언뜻 보기에는 주문 정보를 한눈에 볼 수 있어 편해 보인다. 하지만 이 테이블에는 심각한 문제점들이 숨어있다. 이런 문제들을 '데이터 이상(Anomaly)' 현상이라고 부른다.

- **갱신 이상 (Update Anomaly)**: 만약 '션' 회원의 주소가 '부산시 해운대구'로 변경된다면 어떻게 될까? 이 회원이 주문한 모든 주문 데이터(1001번, 1003번)를 찾아서 주소를 일일이 수정해야 한다. 만약 하나라도 빠뜨린다면 데이터의 일관성이 깨진다.
- **삽입 이상 (Insertion Anomaly)**: 아직 한 번도 주문하지 않은 새로운 회원을 시스템에 등록하고 싶다면? 이 테이블 구조에서는 주문 정보(`order_id`)가 없으면 회원 정보를 추가할 수 없다. 마찬가지로, 아직 아무도 주문하지 않은 새로운 상품을 등록하는 것도 불가능하다.
- **삭제 이상 (Deletion Anomaly)**: 만약 1002번 주문을 취소해서 해당 데이터를 삭제했다고 가정해 보자. 이 주문이 '네이트' 회원의 유일한 주문이었다면, '네이트'라는 회원 정보 자체가 데이터베이스에서 사라져 버린다.

이러한 문제들을 해결하기 위해 데이터베이스 이론가들이 만들어낸 것이 바로 '정규화'다. 정규화는 몇 가지 단계를 거치는데, 각 단계를 '정규형(Normal Form)'이라고 부른다. 이제부터 이 재앙적인 테이블을 차근차근 정규화해 나가 보자.

함수 종속성 (Functional Dependency)

정규화를 이해하기 위한 가장 핵심적인 기초 개념은 '함수 종속성'이다. 어렵게 들리지만 사실은 단순한 내용이다.

함수 종속성이란, 테이블에서 컬럼의 값들이 다른 컬럼의 값을 유일하게 결정하는 관계를 의미한다. 기호로는 `X → Y` 와 같이 표기하며, "**X가 Y를 함수적으로 결정한다**"라고 읽는다. 여기서 X를 **결정자(Determinant)**, Y를 **종속자(Dependent)**라고 한다.

쉽게 말해, X 값이 Y 값을 유일하게 결정한다는 뜻이다.

우리의 예시 테이블에서 함수 종속성을 찾아보자.

- `member_id → member_name` : `member_id`가 1이면 `member_name`은 항상 '션'이다.
- `member_id → member_address` : `member_id`가 1이면 `member_address`는 항상 '서울시 송파구'다.

정규화는 바로 이 함수 종속성 관계를 분석해서, 잘못된 종속 관계를 찾아내고 테이블을 분리하여 올바른 종속 관계로 만들어가는 과정이다.

> 용어
>
> 정규화에서는 다음과 같은 용어를 주로 사용한다.
>
> - 테이블 대신에 릴레이션
> - 행 대신에 튜플
> - 컬럼 대신에 애트리뷰트(속성)

제1 정규형

제1 정규형이 필요한 이유

`disaster_orders` 테이블의 `product_info` 컬럼을 보자. `'10:노트북:2:1500000,15:키보드:1:50000'` 와 같이 여러 상품 정보가 컬럼에 쉼표(,)로 구분되어 들어가 있다. 그리고 각 상품의 상세 정보 들이 `:`로 구분되어 들어가 있다. 이런 방식은 다음과 같은 심각한 문제를 야기한다.

- **데이터 검색의 어려움**: '키보드'가 포함된 주문을 찾으려면 모든 `product_infos` 문자열을 파싱해서 검색해야

한다. 이는 매우 비효율적이며, SQL의 `WHERE` 절을 제대로 활용할 수 없다. 특히 인덱스를 제대로 활용할 수 없다.

- **데이터 수정의 복잡성**: 1001번 주문에서 키보드 수량을 1개에서 2개로 바꾸려면, 이 복잡한 문자열을 애플리케이션에서 읽어와서 분리하고, 수량을 수정한 뒤, 다시 합쳐서 업데이트해야 한다. 데이터베이스가 해야 할 일을 애플리케이션이 떠안게 된다.
- **데이터 타입 사용 불가**: 상품 가격, 수량 등은 숫자 타입으로 관리되어야 합계나 평균 같은 집계 함수를 쉽게 사용할 수 있다. 하지만 문자열 안에 섞여 있으면 불가능하다.

이런 문제를 해결하는 첫 번째 단계가 바로 제1 정규화다.

제1 정규형의 정의와 적용

> **제1 정규형(1NF)**
> 테이블의 모든 컬럼이 **원자적(Atomic)인 값**만을 가져야 한다.

'원자적'이라는 것은 더 이상 쪼갤 수 없는 값을 의미한다. `product_infos` 컬럼은 상품 ID, 상품명, 수량, 가격 등 여러 정보로 쪼갤 수 있으므로 원자적이지 않다.

제1 정규화를 적용하기 위해, 반복되는 상품 정보를 별도의 행으로 분리해야 한다.

단계적 설명 및 예제
먼저, 제1 정규화를 위반하는 테이블을 직접 만들어보자.

```
DROP TABLE IF EXISTS orders_1nf_violating;

-- 제1 정규형을 위반하는 테이블 생성
CREATE TABLE orders_1nf_violating (
    order_id INT NOT NULL,
    ordered_at DATETIME NOT NULL,
    member_id INT NOT NULL,
    member_name VARCHAR(50) NOT NULL,
    product_infos VARCHAR(255) NOT NULL -- 이 컬럼이 원자성을 위반한다.
);

-- 데이터 삽입
```

```
INSERT INTO orders_1nf_violating (order_id, ordered_at, member_id,
member_name, product_infos) VALUES
(1001, '2025-08-20 10:00:00', 1, '션', '10:노트북:2:1500000,15:키보드:1:50000'),
(1002, '2025-08-21 11:00:00', 2, '네이트', '10:노트북:1:1500000'),
(1003, '2025-08-21 12:00:00', 1, '션', '20:마우스:1:30000');
```

실행 결과 확인

```
SELECT * FROM orders_1nf_violating;
```

[실행 결과]

order_id	ordered_at	member_id	member_name	product_infos
1001	2025-08-20 10:00:00	1	션	10:노트북:2:1500000, 15:키보드:1:50000
1002	2025-08-21 11:00:00	2	네이트	10:노트북:1:1500000
1003	2025-08-21 12:00:00	1	션	20:마우스:1:30000

이제 이 테이블을 제1 정규형에 맞게 수정해 보자. `product_infos` 컬럼을 분리하여 각 상품이 별도의 행을 갖도록 만든다.

```
DROP TABLE IF EXISTS orders_1nf;

-- 제1 정규형을 만족하는 테이블 생성
CREATE TABLE orders_1nf (
    order_id       INT NOT NULL,
    member_id      INT NOT NULL,
    member_name    VARCHAR(50) NOT NULL,
    product_id     INT NOT NULL,
    product_name   VARCHAR(100) NOT NULL,
    product_price  INT NOT NULL,
    order_quantity INT NOT NULL,
    ordered_at     DATETIME NOT NULL,
    PRIMARY KEY (order_id, product_id) -- 기본 키를 (order_id, product_id) 복합키로
설정
```

```
);

-- 데이터 삽입
INSERT INTO orders_1nf (order_id, member_id, member_name, product_id,
product_name, product_price, order_quantity, ordered_at) VALUES
(1001, 1, '션', 10, '노트북', 1500000, 2, '2025-08-20 10:00:00'),
(1001, 1, '션', 15, '키보드', 50000, 1, '2025-08-20 10:00:00'),
(1002, 2, '네이트', 10, '노트북', 1500000, 1, '2025-08-21 11:00:00'),
(1003, 1, '션', 20, '마우스', 30000, 1, '2025-08-21 12:00:00');
```

실행 결과 확인

```
SELECT * FROM orders_1nf;
```

[실행 결과]

order_id	member_id	member_name	product_id	product_name	product_price	order_quantity	ordered_at
1001	1	션	10	노트북	1500000	2	2025-08-20 10:00:00
1001	1	션	15	키보드	50000	1	2025-08-20 10:00:00
1002	2	네이트	10	노트북	1500000	1	2025-08-21 11:00:00
1003	1	션	20	마우스	30000	1	2025-08-21 12:00:00

이제 모든 컬럼이 원자적인 값을 갖게 되었다. 하지만 여전히 데이터 중복 문제는 해결되지 않았다. 1001번 주문의 '션' 회원 이름과 주문 날짜가 두 번이나 반복해서 저장되고 있다. 이 문제는 다음 정규형에서 해결하겠다.

제2 정규형

`orders_1nf`를 보니 상품의 가격(`product_price`)은 있는데, 주문 시점의 가격 스냅샷인 주문 가격(`order_price`)이 없다. 주문 가격 필드도 추가하자.

order_id	member_id	member_name	product_id	product_name	product_price	order_price	order_quantity
1001	1	션	10	노트북	1500000	1500000	2
1001	1	션	15	키보드	50000	50000	1
1002	2	네이트	10	노트북	1500000	1500000	1
1003	1	션	20	마우스	30000	30000	1

- 표 길이가 너무 길어서 `ordered_at` 컬럼 생략

제2 정규형이 필요한 이유

`orders_1nf` 테이블을 다시 살펴보자. 이 테이블의 기본 키(PK)는 `(order_id, product_id)` 복합키다. 즉, 주문 ID와 상품 ID가 합쳐져야 하나의 행을 고유하게 식별할 수 있다.

여기서 함수 종속 관계를 분석해 보자.

기본 키 모두에 종속되는 컬럼들

1. `(order_id, product_id) → order_price`
2. `(order_id, product_id) → order_quantity`
    - `order_price`와 `order_quantity`는 기본 키 전체에 종속된다. 올바른 관계다.

기본 키의 일부에만 종속되는 컬럼들

1. `order_id → member_id`
2. `order_id → member_name`
3. `order_id → ordered_at`
    - `member_id`, `member_name`, `ordered_at`는 기본 키의 일부인 `order_id`에만 종속된다. `product_id`와는 아무런 관련이 없다.
4. `product_id → product_name`, `product_price`
    - `product_name`, `product_price`는 기본 키의 일부인 `product_id`에만 종속된다.

이처럼, **기본 키의 일부에만 종속되는 컬럼이 존재하는 것**을 **부분 함수 종속(Partial Functional Dependency)**이라고 한다.
부분 함수 종속은 다음과 같은 데이터 중복 및 이상 현상을 일으킨다.

- **데이터 중복**: 1001번 주문에 상품이 10개 포함된다면, `member_name`('션')과 `ordered_at`는 10번 반복 저장되어야 한다. 이는 심각한 공간 낭비다.
- **갱신 이상**: '션' 회원이 개명이라도 하면, 관련된 모든 주문-상품 데이터를 찾아 이름을 수정해야 한다.
- **삽입/삭제 이상**: 제1 정규형에서 보았던 이상 현상이 여전히 일부 남아있다.

제2 정규형의 정의와 적용

> **제2 정규형(2NF)**
> 1. 제1 정규형을 만족해야 한다.
> 2. 테이블의 모든 컬럼이 기본 키에 대해 **완전 함수 종속(Fully Functional Dependent)**이어야 한다.
>   - 즉, 부분 함수 종속이 없어야 한다

부분 함수 종속을 제거하는 방법은 간단하다. **종속 관계에 맞게 테이블을 분리**하면 된다.

- `order_id`에만 종속되는 컬럼들 (`member_id`, `member_name`, `ordered_at`)을 모아 `orders` 테이블을 만든다.
- `product_id`에만 종속되는 컬럼들 (`product_name`, `product_price`)을 모아 `product` 테이블을 만든다.
- (`order_id`, `product_id`) 기본 키 전체에 종속되는 컬럼들 (`order_quantity`, `order_price`)을 모아 연결 테이블인 `order_item`을 만든다.

단계적 설명 및 예제

`orders_1nf` 테이블을 3개의 테이블로 분리해 보자.

```
DROP TABLE IF EXISTS order_item_2nf;
DROP TABLE IF EXISTS orders_2nf;
DROP TABLE IF EXISTS product_2nf;
```

```sql
-- 1. orders 테이블 생성 (주문 정보)
CREATE TABLE orders_2nf (
    order_id   INT NOT NULL,
    member_id  INT NOT NULL,
    member_name VARCHAR(50) NOT NULL, -- 아직 3NF 위반 요소를 남겨둔다.
    ordered_at DATETIME NOT NULL,
    PRIMARY KEY (order_id)
);

-- 2. product 테이블 생성 (상품 정보)
CREATE TABLE product_2nf (
    product_id   INT NOT NULL,
    product_name VARCHAR(100) NOT NULL,
    product_price        INT NOT NULL,
    PRIMARY KEY (product_id)
);

-- 3. order_item 테이블 생성 (주문-상품 연결 정보)
CREATE TABLE order_item_2nf (
    order_id     INT NOT NULL,
    product_id   INT NOT NULL,
    order_price  INT NOT NULL,
    order_quantity        INT NOT NULL,
    PRIMARY KEY (order_id, product_id),
    FOREIGN KEY (order_id) REFERENCES orders_2nf (order_id),
    FOREIGN KEY (product_id) REFERENCES product_2nf (product_id)
);
```

분리된 테이블에 맞추어 데이터를 삽입한다.

```sql
-- product_2nf 데이터 삽입
INSERT INTO product_2nf (product_id, product_name, product_price) VALUES
(10, '노트북', 1500000),
(15, '키보드', 50000),
(20, '마우스', 30000);

-- orders_2nf 데이터 삽입
INSERT INTO orders_2nf (order_id, member_id, member_name, ordered_at) VALUES
(1001, 1, '션', '2025-08-20 10:00:00'),
(1002, 2, '네이트', '2025-08-21 11:00:00'),
```

```
(1003, 1, '션', '2025-08-21 12:00:00');

-- order_item_2nf 데이터 삽입
INSERT INTO order_item_2nf (order_id, product_id, order_price, order_quantity)
VALUES
(1001, 10, 1500000, 2),
(1001, 15, 50000, 1),
(1002, 10, 1500000, 1),
(1003, 20, 30000, 1);
```

실행 결과 확인

```
SELECT * FROM orders_2nf;
```

[실행 결과]

order_id	member_id	member_name	ordered_at
1001	1	션	2025-08-20 10:00:00
1002	2	네이트	2025-08-21 11:00:00
1003	1	션	2025-08-21 12:00:00

```
SELECT * FROM product_2nf;
```

[실행 결과]

product_id	product_name	price
10	노트북	1500000
15	키보드	50000
20	마우스	30000

```
SELECT * FROM order_item_2nf;
```

[실행 결과]

order_id	product_id	order_price	order_quantity
1001	10	1500000	2
1001	15	50000	1
1002	10	1500000	1
1003	20	30000	1

이제 부분 함수 종속이 사라졌다. `orders_2nf` 테이블의 데이터 중복(주문 날짜, 회원 이름)도 `orders_1nf` 테이블에 비해 훨씬 줄어들었다. 하지만 `orders_2nf` 테이블을 자세히 보면, 여전히 갱신 이상의 문제가 남아있다. '션' 회원의 이름이 두 번이나 중복 저장되어 있다. 이 문제를 해결하는 것이 제3 정규화다.

제3 정규형

제3 정규형이 필요한 이유

`orders_2nf` 테이블을 분석해 보자. 기본 키는 `order_id`다.

order_id	member_id	member_name	ordered_at
1001	1	션	2025-08-20 10:00:00
1002	2	네이트	2025-08-21 11:00:00
1003	1	션	2025-08-21 12:00:00

함수 종속 관계 분석

`member_id`와 `ordered_at`는 기본 키인 `order_id`에 잘 종속되어 있다.

- `order_id → member_id`

- `order_id → ordered_at`

그런데 `member_name`은 기본 키가 아닌 `member_id`에 종속된다.

- `member_id → member_name`

즉, `order_id → member_id → member_name` 과 같은 종속 관계가 나타난다. 이처럼 **기본 키가 아닌 컬럼이 다른 컬럼을 결정하는 관계**를 **이행적 함수 종속(Transitive Functional Dependency)**이라고 한다.

> **이행적(Transitive) 단어**
> '이행적(Transitive)'이라는 단어는 '거쳐서 간다', '옮겨 간다'는 의미를 가지고 있다.

데이터베이스에서는 A → B 이고 B → C 일 때, 결과적으로 A → C가 성립하는 관계를 생각하면 이해하기 쉽다. 여기서 A가 C를 결정하는 관계는 B를 **거쳐서 가는(이행적인)** 관계가 되는 것이다. 여기서 A는 일반적으로 테이블의 기본 키이다.

우리 예시에서는 `order_id`가 `member_name`을 결정하지만, 중간에 `member_id`를 거쳐서 간다.

- `order_id` → `member_id` (주문 ID를 알면 회원 ID를 알 수 있다.)
- `member_id` → `member_name` (회원 ID를 알면 회원 이름을 알 수 있다.)

따라서 `order_id` → `member_name` 관계는 `member_id`라는 일반 컬럼을 거쳐가는 **이행적 함수 종속** 관계가 된다. 제3 정규형은 바로 이 이행적 함수 종속을 제거하는 과정이다.

이행적 함수 종속은 다음과 같은 문제를 일으킨다.

- **갱신 이상**: '션' 회원이 '김개발'로 개명하면, `member_id`가 1인 모든 주문을 찾아 이름을 변경해야 한다. 누락되면 데이터 불일치가 발생한다.
- **삽입 이상**: 아직 주문을 한 번도 하지 않은 신규 회원은 `orders` 테이블에 등록할 수 없다. `order_id`가 없기 때문이다.
- **삭제 이상**: 1002번 주문을 삭제했는데, 이 주문이 '네이트' 회원의 유일한 주문이었다면 '네이트' 회원 정보 자체가 사라진다.

제3 정규형의 정의와 적용

> **제3 정규형(3NF)**
>
> 1. 제2 정규형을 만족해야 한다.
> 2. 이행적 함수 종속이 존재하지 않아야 한다.

이행적 함수 종속을 제거하는 방법 역시 테이블을 분리하는 것이다. 결정자(`member_id`)와 종속자(`member_name`)를 묶어 별도의 `member` 테이블로 분리하면 된다.

단계적 설명 및 예제

`orders_2nf` 테이블을 `orders_3nf`와 `member_3nf` 테이블로 분리하자.

```
DROP TABLE IF EXISTS orders_3nf;
DROP TABLE IF EXISTS member_3nf;

-- 1. member 테이블 생성 (회원 정보)
CREATE TABLE member_3nf (
    member_id INT NOT NULL,
    member_name VARCHAR(50) NOT NULL,
    PRIMARY KEY (member_id)
);

-- 2. orders 테이블 생성 (주문 정보, member_name 제거)
CREATE TABLE orders_3nf (
    order_id   INT NOT NULL,
    member_id  INT NOT NULL,
    ordered_at DATETIME NOT NULL,
    PRIMARY KEY (order_id),
    FOREIGN KEY (member_id) REFERENCES member_3nf (member_id)
);
```

분리된 테이블에 데이터를 삽입한다.

```
-- member_3nf 데이터 삽입 (회원 정보는 이제 중복 없이 한 번만 저장된다)
INSERT INTO member_3nf (member_id, member_name) VALUES
(1, '션'),
(2, '네이트');

-- orders_3nf 데이터 삽입
```

```sql
INSERT INTO orders_3nf (order_id, member_id, ordered_at) VALUES
(1001, 1, '2025-08-20 10:00:00'),
(1002, 2, '2025-08-21 11:00:00'),
(1003, 1, '2025-08-21 12:00:00');
```

실행 결과 확인

```sql
SELECT * FROM member_3nf;
```

[실행 결과]

member_id	member_name
1	션
2	네이트

```sql
SELECT * FROM orders_3nf;
```

[실행 결과]

order_id	member_id	ordered_at
1001	1	2025-08-20 10:00:00
1002	2	2025-08-21 11:00:00
1003	1	2025-08-21 12:00:00

이제 `member_name`은 `member_3nf` 테이블에 단 한 번만 저장된다. 만약 '션' 회원의 이름이 바뀌면 `member_3nf` 테이블의 단 하나의 행만 수정하면 된다. 그러면 이 회원을 참조하는 모든 주문에서 변경된 이름 정보를 일관성 있게 조회할 수 있다.

이제 데이터 이상 현상이 모두 해결되었다.

제3 정규형 심화: 후보 키와 이행 종속

> **"제3 정규형 심화: 후보 키와 이행 종속" 내용은 촬영 이후에 메뉴얼에 추가된 부분입니다.**

여기서 실무에서 많이 헷갈려 하는 부분을 하나 짚고 넘어가겠다.
`member` 테이블을 생각해 보자. (`member_id` PK, `login_id` UK, `password`, `email` UK)

이 테이블에는 다음과 같은 종속 관계가 있다.

- `member_id → login_id`
- `login_id → password`, `email`

이 관계가 `member_id → login_id → password` 처럼 보여서 이행적 함수 종속으로 오해할 수 있다.

결론부터 말하면, **이것은 제3 정규형 위반이 아니다.**

"왜?"
제3 정규형이 막으려는 진짜 문제는, **'키가 아닌 일반 속성'**이 **'다른 일반 속성'**을 결정하는 상황이다.
앞선 `orders_2nf` 예시(`order_id → member_id → member_name`)에서는 `member_id`가 `orders_2nf` 테이블의 키가 아닌 '일반 속성'이면서 `member_name`이라는 '일반 속성'을 결정했기 때문에 문제가 되었다.

하지만 `member` 테이블의 `login_id`는 '일반 속성'이 아니다.

후보 키 (Candidate Key)
데이터베이스에는 행을 유일하게 식별할 수 있는 여러 '키'들이 있다.

- **후보 키 (Candidate Key)**: 튜플(행)을 유일하게 식별할 수 있는 속성.
- **기본 키 (Primary Key)**: 여러 후보 키 중 대표로 선택한 키.

`member` 테이블에서 `member_id`, `login_id`, `email`은 모두 `UNIQUE`하므로 행을 유일하게 식별할 수 있다. 따라서 이 셋은 모두 **후보 키**다.

`login_id`는 '일반 속성'이 아니라 당당한 '후보 키' 자격을 가지고 있다.
키(후보 키)가 다른 속성을 결정하는 것은 지극히 정상적이고 바람직한 관계다.

따라서 `login_id → password` 관계는 후보 키가 일반 속성을 결정하는 것이므로, 제3 정규형에 전혀 위배되지 않는다. 이 테이블은 이미 제3 정규형을 만족하는, 잘 설계된 테이블이다.

BCNF 정규형

BCNF(Boyce-Codd Normal Form)는 제3 정규형을 조금 더 강화한 버전으로, '강한 제3 정규형'이라고도 불린다. 실무에서는 제3 정규형까지 만족하는 설계를 목표로 하는 경우가 많지만, BCNF를 이해하면 더 깊이 있는 데이터베이스 설계가 가능하다.

> **BCNF (Boyce-Codd Normal Form)**
> 테이블의 모든 **결정자(Determinant)**가 **후보 키(Candidate Key)**여야 한다.

용어가 조금 어렵게 들릴 수 있다.

- **결정자**는 함수 종속 관계(`X → Y`)에서 `X`에 해당하는, 즉 다른 컬럼의 값을 결정하는 컬럼(또는 컬럼의 집합)을 의미한다.
- **후보 키**는 테이블의 행을 유일하게 식별할 수 있는 컬럼의 최소 집합이다.
- **기본 키**는 여러 후보 키 중 하나를 선택한 것이다.

제3 정규형은 기본 키가 아닌 컬럼들 사이의 종속 관계(이행적 종속)만 제거했다. 반면 BCNF는 더 엄격해서, 기본 키가 아니더라도 어떤 컬럼이 다른 컬럼을 결정한다면, 그 결정자는 반드시 후보 키여야 한다는 규칙이다.

대부분의 경우 제3 정규형을 만족하면 BCNF도 만족한다. 하지만 아주 드물게 제3 정규형은 만족하지만 BCNF는 만족하지 않는 경우가 있다. 지금부터 그 특별한 경우를 예제를 통해 알아보자.

BCNF가 필요한 이유: 3NF의 한계

BCNF가 실무에서는 자주 발생하지 않기 때문에, BCNF를 설명할 때 자주 사용하는 전통적인 학교 수강 신청 예시를 사용하겠다.
어떤 학교의 특강 수강신청 시스템을 설계한다고 가정해 보자. 다음과 같은 비즈니스 규칙이 있다.

1. 한 학생은 여러 개의 특강을 신청할 수 있다.
2. 하나의 특강은 여러 명의 교수가 가르칠 수 있다. (예: '데이터베이스' 특강을 김 교수, 박 교수가 함께 진행)
3. **한 교수는 오직 하나의 특강만 담당한다.** (이 규칙이 BCNF 위반을 만드는 핵심이다)

이 규칙에 따라 다음과 같은 `special_lecture` 테이블을 설계했다.

student_id	lecture_name	professor_name
101	데이터베이스	김교수
101	자바	서교수
102	데이터베이스	박교수
103	자바	서교수

함수 종속성 분석

1. `{student_id, lecture_name} → professor_name`: 학생 ID와 특강 이름을 알면 담당 교수를 알 수 있다.
2. `professor_name → lecture_name`: 교수의 이름을 알면 그가 담당하는 특강 이름을 알 수 있다. (규칙 3번)

후보 키 분석

- `{student_id, lecture_name}`은 모든 컬럼을 결정하므로 후보 키가 될 수 있다.
- `{student_id, professor_name}`은 어떨까? 학생 ID와 교수 이름을 알면 `lecture_name`도 알 수 있으므로(`professor_name → lecture_name`), 이 조합 역시 후보 키가 될 수 있다.

이 테이블에는 두 개의 후보 키가 존재한다. 우리는 `(student_id, lecture_name)`을 기본 키(PK)로 선택했다고 가정하자.

정규형 만족 여부 확인

- **제1 정규형**: 모든 컬럼은 원자적 값을 가지므로 만족한다.
- **제2 정규형**: 기본 키는 `(student_id, lecture_name)`이다. 기본 키가 아닌 컬럼은 `professor_name` 하나뿐이며, 이 컬럼은 기본 키 전체에 종속되므로 부분 함수 종속이 없다. 만족한다.
- **제3 정규형**: 이행적 함수 종속이 있는가? 기본 키가 아닌 컬럼이 다른 기본 키가 아닌 컬럼을 결정해야 한다. 하지만 이 테이블에는 기본 키가 아닌 컬럼이 `professor_name` 하나뿐이므로, 이행적 종속이 발생할 수 없다. 만족한다.
- **BCNF**: 모든 결정자가 후보 키인가?
  - `{student_id, lecture_name}`는 `professor_name`을 결정한다. 이 결정자는 기본 키이므로 후

보 키다. (OK)

- `professor_name`은 `lecture_name`을 결정한다. **이 결정자(`professor_name`)는 후보 키인가?**
  아니다.
  - `professor_name`만으로는 행을 유일하게 식별할 수 없기 때문에 후보 키가 아니다.
  - 따라서 **BCNF를 위반한다.**

BCNF를 위반했을 때의 문제점

제3 정규형까지 만족했는데, 왜 문제가 될까? `professor_name`이 `lecture_name`을 결정하는 종속성 때문에 여전히 데이터 이상 현상이 발생한다.

- **갱신 이상 (Update Anomaly)**: 만약 '서교수'가 담당하는 특강이 '자바'에서 '파이썬'으로 변경된다면, '서교수'가 등장하는 모든 행을 찾아 `lecture_name`을 '파이썬'으로 바꿔야 한다. 하나라도 누락하면 '서교수'가 '자바'와 '파이썬'을 모두 가르치는 것처럼 보이는 데이터 불일치가 발생한다.
- **삽입 이상 (Insertion Anomaly)**: '최교수'가 '인공지능' 특강을 새로 개설했지만 아직 신청한 학생이 없다면, 이 정보를 테이블에 추가할 수 없다. 기본 키인 `student_id`가 `NULL`이 될 수 없기 때문이다.
- **삭제 이상 (Deletion Anomaly)**: 만약 102번 학생이 '데이터베이스' 수강을 취소하여 해당 행이 삭제된다고 가정하자. 이로 인해 '박교수'가 '데이터베이스'를 담당한다는 소중한 정보가 데이터베이스에서 완전히 사라져 버린다.

BCNF 적용: 테이블 분리

BCNF를 만족시키기 위한 해결책은 정규화의 기본 원칙과 같다. 문제가 되는 함수 종속성을 기준으로 테이블을 분리하는 것이다. 문제가 되는 종속성은 `professor_name → lecture_name`이다.

이 결정자(`professor_name`)와 종속자(`lecture_name`)를 묶어 새로운 테이블로 분리한다.

단계적 설명 및 예제

먼저, BCNF를 위반하는 테이블을 생성하고 데이터를 넣어보자.

```
DROP TABLE IF EXISTS special_lecture_bcnf_violating;

-- BCNF를 위반하는 테이블 생성
CREATE TABLE special_lecture_bcnf_violating (
    student_id     INT NOT NULL,
    lecture_name   VARCHAR(50) NOT NULL,
```

```
    professor_name VARCHAR(50) NOT NULL,
    PRIMARY KEY (student_id, lecture_name)
);

-- 데이터 삽입
INSERT INTO special_lecture_bcnf_violating (student_id, lecture_name,
professor_name) VALUES
(101, '데이터베이스', '김교수'),
(101, '자바', '서교수'),
(102, '데이터베이스', '박교수'),
(103, '자바', '서교수');
```

실행 결과 확인

```
SELECT * FROM special_lecture_bcnf_violating;
```

[실행 결과]

student_id	lecture_name	professor_name
101	데이터베이스	김교수
101	자바	서교수
102	데이터베이스	박교수
103	자바	서교수

이제 이 테이블을 BCNF에 맞게 두 개로 분리한다.

```
DROP TABLE IF EXISTS enrollment_bcnf;
DROP TABLE IF EXISTS professor_bcnf;

-- 1. professor 테이블 생성 (교수와 담당 특강 정보)
CREATE TABLE professor_bcnf (
    professor_name VARCHAR(50) NOT NULL,
    lecture_name   VARCHAR(50) NOT NULL,
    PRIMARY KEY (professor_name)
```

```
);

-- 2. enrollment 테이블 생성 (수강 신청 정보)
CREATE TABLE enrollment_bcnf (
    student_id     INT NOT NULL,
    professor_name VARCHAR(50) NOT NULL,
    PRIMARY KEY (student_id, professor_name),
    FOREIGN KEY (professor_name) REFERENCES professor_bcnf (professor_name)
);
```

분리된 테이블에 데이터를 다시 정리해서 삽입한다.

```
-- professor_bcnf 데이터 삽입 (교수-특강 정보는 중복 없이 저장된다)
INSERT INTO professor_bcnf (professor_name, lecture_name) VALUES
('김교수', '데이터베이스'),
('서교수', '자바'),
('박교수', '데이터베이스');

-- enrollment_bcnf 데이터 삽입
INSERT INTO enrollment_bcnf (student_id, professor_name) VALUES
(101, '김교수'),
(101, '서교수'),
(102, '박교수'),
(103, '서교수');
```

실행 결과 확인

```
SELECT * FROM professor_bcnf;
```

[실행 결과]

professor_name	lecture_name
김교수	데이터베이스
박교수	데이터베이스
서교수	자바

```sql
SELECT * FROM enrollment_bcnf;
```

[실행 결과]

student_id	professor_name
101	김교수
101	서교수
102	박교수
103	서교수

이제 모든 데이터 이상 현상이 해결되었다.

- **갱신**: '서교수'의 담당 과목이 바뀌면 `professor_bcnf` 테이블의 단 한 행만 수정하면 된다.
- **삽입**: 아직 신청 학생이 없는 '최교수'의 '인공지능' 특강 정보도 `professor_bcnf` 테이블에 자유롭게 추가할 수 있다.
- **삭제**: 102번 학생이 수강을 취소해서 `enrollment_bcnf` 테이블의 데이터가 삭제되어도, '박교수'가 '데이터베이스'를 담당한다는 정보는 `professor_bcnf` 테이블에 안전하게 남아있다.

이렇게 BCNF는 제3 정규형이 놓칠 수 있는 복잡한 종속 관계에서 발생하는 데이터 이상 현상을 해결해 준다. 실무에서 BCNF 위반 사례는 흔치 않지만, 여러 개의 후보 키가 존재하고 그 후보 키들이 서로의 일부를 포함하는 복잡한 관계가 나타날 때 BCNF를 검토해 보면 더 견고한 설계를 할 수 있다.

실무와 정규화

정규화를 사용하는 이유를 한번 정리해보자.

- **데이터 중복 최소화**: 불필요한 데이터 중복을 제거하여 저장 공간을 효율적으로 사용한다.
- **데이터 일관성 및 무결성 확보**: 데이터가 여러 곳에 중복 저장되지 않으므로, 수정 시 발생할 수 있는 데이터 불일

치 가능성을 원천적으로 차단한다.
- **데이터 이상 현상(Anomaly) 해결**: 삽입, 갱신, 삭제 이상 현상을 방지하여 데이터의 신뢰성을 높인다.
- **유연한 데이터 구조**: 새로운 데이터 요구사항이 발생했을 때, 테이블 구조 변경을 최소화하며 시스템을 확장할 수 있게 한다.

정규화의 결과

우리가 처음에 봤던 재앙적인 `disaster_orders` 테이블에서 시작해서, 1NF, 2NF, 3NF를 거쳐 최종적으로 어떤 테이블들을 얻었는지 정리해 보자.

1. `member_3nf` : 회원 정보를 저장한다.
2. `product_2nf` : 상품 정보를 저장한다.
3. `orders_3nf` : 주문 기본 정보를 저장하며, 회원을 참조한다.
4. `order_item_2nf` : 주문과 상품을 연결하며, 각 상품의 주문 수량과 가격을 저장한다.

이 구조를 어디서 많이 보지 않았는가? 바로 이 강의의 맨 처음에 제시되었던, 우리가 최종 목표로 삼았던 테이블 스키마와 정확히 일치한다.

- `member_3nf` → `member` 테이블
- `product_2nf` → `product` 테이블
- `orders_3nf` → `orders` 테이블
- `order_item_2nf` → `order_item` 테이블

결국, 우리가 처음부터 잘 설계된 테이블 구조를 제시했던 것은 바로 이 **정규화 원칙을 이미 적용한 결과물**이었던 것이다.

정규화가 왜 필요할까?

아마 이런 의문이 들 수 있다. '정규화 이론을 배우기 전이었는데, 어떻게 우리는 자연스럽게 정규화된 테이블을 생각할 수 있었을까?'

그 이유는 사실 우리의 직관적인 사고방식이 데이터 정규화의 목표와 매우 유사하기 때문이다. 우리는 현실 세계의 사물이나 개념을 인식할 때, 자연스럽게 다음과 같은 원칙을 적용한다.

- **'하나의 칸에는 하나의 정보만'**: 하나의 주문에 여러 상품이 포함될 때, 상품 정보를 한 칸에 `노트북, 키보드, 마우스` 처럼 몰아넣는 것은 나중에 관리하기 어렵다는 것을 쉽게 예상할 수 있다. 각 주문 상품을 별도의 행으로 구분하여 관리하는 것이 훨씬 깔끔하고 명확하다. 이것이 바로 제1정규형, 즉 '원자성'의 원칙이다.

- **'하나의 주제는 하나의 묶음으로'**: 우리는 '회원'에 대한 정보(이름, 주소, 이메일)와 '상품'에 대한 정보(상품명, 가격, 재고)가 서로 다른 주제라는 것을 본능적으로 안다. 그래서 자연스럽게 `member` 테이블과 `product` 테이블을 분리해서 생각한다. 이것이 바로 제2정규형과 제3정규형의 핵심 아이디어인 '하나의 기본 키는 하나의 주제에 대한 사실만을 설명해야 한다'는 원칙과 맞닿아 있다.

- **'반복되는 정보는 한 곳에서 관리'**: '션' 회원이 주문을 100번 하더라도, 그의 이름과 주소를 100번 저장하는 것은 비효율적이라고 직관적으로 느낀다. 대신 '션'이라는 회원 정보를 한 곳(`member` 테이블)에만 저장하고, 주문할 때는 그 회원을 가리키는 '표시(`member_id`)'만 사용하는 것이 합리적이다. 이 방식은 데이터 중복을 피하고 갱신 이상의 문제를 해결하는데, 이는 정확히 제3 정규화가 달성하려는 목표다.

이처럼, 상식적이고 합리적으로 데이터를 정리하려는 노력은 자연스럽게 정규화된 형태로 귀결되는 경우가 많다.

그렇다면 정규화 이론은 왜 배워야 할까? 정규화는 우리의 직관이 맞았는지 검증할 수 있는 **체계적인 도구**를 제공한다. 또한, BCNF 예시처럼 우리의 직관만으로는 파악하기 어려운 복잡한 데이터 종속 관계를 발견하고 해결할 수 있는 **이론적 기반**이 되어준다. 즉, 정규화는 우리의 '감'을 '논리'로 바꾸어 주는 과정이다.

실무의 전문가들은 정규화를 어떻게 사용하는가?

훌륭한 요리사는 매번 요리책의 레시피를 순서대로 외워서 따라하지 않는다. 훌륭한 요리사는 재료의 본질과 조리 과정의 원리를 이해하기에 레시피 없이도 훌륭한 요리를 만들어낸다.

데이터베이스 전문가들도 마찬가지다. 그들은 '1정규형을 적용하고, 다음은 2정규형...'과 같은 방식으로 생각하지 않는다. 대신, 다루어야 할 데이터의 본질을 파악하고 '회원', '상품', '주문'과 같은 핵심 개념(엔티티)들을 식별한다. 그리고 각 개념에 맞는 데이터들을 자연스럽게 묶어준다.

이처럼 **개념을 중심으로 데이터를 모델링하는 사고방식** 자체가 이미 정규화의 원칙을 내포하고 있다. 전문가에게 정규화 이론은 매번 따라야 하는 규칙 목록이라기보다는, 자신의 직관적인 설계가 논리적으로 탄탄한지 검증하고, 다른 개발자들과 명확하게 소통하기 위한 **언어이자 도구**에 가깝다. 즉, 전문가는 정규화 규칙을 '암기'하는 것이 아니라, 수많은 경험을 통해 그 원칙을 '체화'하고 있는 것이다.

실무 경험 기반 설명

실무에서는 보통 **제3 정규형 또는 BCNF까지 정규화를 진행하는 것을 목표**로 한다. 하지만 정규화가 항상 정답인 것은 아니다. 정규화를 진행하면 테이블이 잘게 쪼개지기 때문에, 원하는 데이터를 얻기 위해 여러 테이블을 `JOIN`해야 하는 경우가 많아진다. 이는 때로 **조회 성능 저하**를 유발할 수 있다.

따라서, 읽기(SELECT) 작업이 매우 빈번하고 성능이 아주 중요한 일부 시스템에서는, 데이터 중복을 감수하고 `JOIN`의 수를 줄이기 위해 의도적으로 정규화 원칙을 위배하는 **역정규화(Denormalization)**를 수행하기도 한다. 역정규화는 뒤에 물리적 모델링에서 자세히 알아본다.

중요한 것은 정규화의 원칙을 명확히 이해하고, 우리가 설계하는 시스템의 특성(데이터의 양, 트랜잭션의 종류, 성능 요구사항 등)에 맞게 정규화와 역정규화 사이에서 최적의 균형점을 찾는 것이다. 이것이 바로 뛰어난 데이터베이스 설계자의 역량이다

> **제4, 제5 정규형**
> 정규형은 4NF(다치 종속성(Multi-valued Dependency)), 5NF(조인 종속성(Join Dependency))까지 계속 이어진다. 하지만 실무에서 3NF나 BCNF 이상의 정규화를 마주하는 경우는 극히 드물다. 따라서 실무에서는 BCNF까지 이해하면 충분하다.

정리

정규화 - 시작

- 정규화는 데이터의 중복을 최소화하고 일관성을 보장하여, 잘 설계된 데이터베이스를 만들기 위한 체계적인 절차이다.
- 하나의 거대한 테이블은 갱신, 삽입, 삭제 이상(Anomaly)과 같은 심각한 문제를 일으킨다.
- 정규화의 핵심 기초 개념은 '함수 종속성'이며, 이는 한 컬럼(결정자)이 다른 컬럼(종속자)의 값을 유일하게 결정하는 관계를 의미한다 (`X → Y`).

제1 정규형

- 제1 정규형(1NF)은 테이블의 모든 컬럼이 더 이상 쪼갤 수 없는 원자적(Atomic)인 값만을 가져야 한다는 규칙이다.
- 한 컬럼에 여러 정보를 쉼표 등으로 구분해 저장하는 방식은 데이터 검색, 수정, 타입 활용을 어렵게 하므로, 각 정보를 별도의 행으로 분리하여 1NF를 만족시킨다.

제2 정규형

- 제2 정규형(2NF)은 제1 정규형을 만족하고, 모든 컬럼이 기본 키 전체에 완전 함수 종속되어야 한다는 규칙이다.
- 복합 기본 키의 일부에만 종속되는 '부분 함수 종속'이 존재해서는 안 된다.

- 부분 함수 종속이 발견되면, 종속 관계에 맞게 테이블을 분리하여 해결한다.

제3 정규형

- 제3 정규형(3NF)은 제2 정규형을 만족하고, 이행적 함수 종속이 없어야 한다는 규칙이다.
- 이행적 함수 종속은 기본 키가 아닌 컬럼이 다른 컬럼을 결정하는 관계(`PK → 일반컬럼1 → 일반컬럼2`)를 의미한다.
- 이행적 종속 관계를 별도의 테이블로 분리하여 데이터 중복과 이상 현상을 제거한다.

BCNF 정규형

- BCNF(Boyce-Codd Normal Form)는 제3 정규형보다 강화된 정규형으로, 테이블의 모든 결정자가 후보 키여야 한다는 규칙이다.
- 제3 정규형을 만족하더라도, 후보 키가 아닌 컬럼이 다른 컬럼을 결정하는 드문 경우에 발생하는 데이터 이상 현상을 해결한다.
- 문제가 되는 함수 종속성을 기준으로 테이블을 분리하여 BCNF를 만족시킨다.

실무와 정규화

- 정규화는 데이터 중복 최소화, 데이터 일관성 확보, 이상 현상 해결, 유연한 구조 확보를 위해 필요하다.
- 상식적이고 합리적으로 데이터를 정리하려는 노력은 자연스럽게 정규화된 형태로 귀결되며, 정규화 이론은 이를 검증하는 체계적인 도구를 제공한다.
- 실무에서는 보통 제3 정규형이나 BCNF를 목표로 하지만, 조회 성능이 매우 중요한 경우 `JOIN`을 줄이기 위해 의도적으로 정규화를 위반하는 '역정규화'를 수행하기도 한다.
- 뛰어난 설계는 시스템의 특성에 맞게 정규화와 역정규화 사이의 최적의 균형점을 찾는 것이다.