2. 설계의 중요성

#0.강의/2.데이터베이스로드맵/3.설계1

데이터베이스 설계의 첫걸음

우리는 지난 데이터베이스 입문편, 기본편을 통해 데이터베이스를 만들고, SQL을 사용해서 데이터를 저장하고 조회하는 기본적인 방법을 모두 익혔다. SELECT, INSERT 같은 기본 문법부터 시작해서 JOIN, 서브쿼리, 인덱스와 트랜잭션에 이르기까지, 데이터를 다루는 데 필요한 핵심적인 기술들을 학습했다.

그런데 왜 '설계'라는 것을 지금부터 따로 시간을 들여 배워야 할까? 그냥 필요한 데이터가 생길 때마다 `CREATE TABLE`로 테이블을 만들고, `ALTER TABLE`로 컬럼을 추가하면 되지 않을까?

이 질문에 답하기 위해 집을 짓는 과정을 한번 생각해보자. 튼튼하고 좋은 집을 짓기 위해 가장 먼저 해야 할 일은 무엇일까? 바로 '설계도'를 그리는 일이다. 어떤 방을 어디에 배치하고, 기둥은 어디에 세울지, 전기 배선과 수도관은 어떻게 연결할지 미리 계획해야 한다. 만약 설계도 없이 감으로만 집을 짓기 시작한다면, 처음에는 그럴듯해 보일지 몰라도 나중에는 벽에 금이 가거나, 문이 제대로 닫히지 않거나, 심지어는 집 전체가 무너져 내리는 끔찍한 결과를 맞이할 수 있다.

데이터베이스 설계도 이와 똑같다. '일단 만들고 보자'는 생각으로 주먹구구식으로 테이블을 만들다 보면, 처음에는 빠르게 개발되는 것처럼 느껴질 수 있다. 하지만 서비스가 커지고 데이터가 복잡해질수록 문제는 눈덩이처럼 불어난다. 데이터가 중복 저장되어 서로 맞지 않게 되거나, 간단한 기능을 하나 추가하기 위해 수십 개의 파일을 수정해야 하거나, 무엇보다 애플리케이션의 속도가 점점 느려져 사용자들이 떠나가는 '재앙'을 맞이하게 된다. 이처럼 설계 없이 진행된 개발은 시간이 지날수록 '기술 부채(Technical Debt)'로 이어진다.

이번 강의에서는 바로 이런 재앙을 막고, 튼튼하고 유연하며 오래가는 데이터베이스 시스템을 만드는 방법을 배울 것이다. 감에 의존하지 않고, 체계적인 원칙과 절차에 따라 데이터의 구조를 잡아나가는 훈련을 한다.

이 강의는 다음과 같은 흐름으로 진행된다.

1. **설계의 중요성**: 먼저 설계가 왜 중요한지, 잘못된 설계가 어떤 재앙을 불러오는지 현실적인 사례를 통해 체감한다.

2. **설계의 3단계 로드맵**: 아이디어를 실제 데이터베이스로 만들어가는 전체 과정(개념적 설계 ➔ 논리적 설계 ➔ 물리적 설계)의 큰 그림을 이해한다.
3. **개념적/논리적 모델링**: 요구 사항을 분석해서 핵심 데이터를 찾아내고(엔티티), 그 데이터들이 가지는 속성을 정의하며, 데이터들 간의 관계를 명확하게 표현하는 방법을 배운다. ERD(Entity-Relationship Diagram) 작성법, 다양한 키(Key)의 종류와 역할, 관계의 종류(1:N, M:N 등)와 식별 및 비식별 관계 등을 깊이 있게 다룬다.
4. **정규화**: 데이터의 중복을 제거하고 무결성을 높여 모델의 품질을 한 단계 끌어올리는 '정규화' 과정을 배운다.
5. **물리적 모델링**: 최종적으로 완성된 논리 모델을 실제 MySQL 데이터베이스에 최적화된 테이블로 만드는 방법을 배운다.

이 모든 과정을 '성공하는 쇼핑몰'이라는 하나의 예제를 가지고 처음부터 끝까지 직접 진행하며, 이론이 실무에서 어떻게 적용되는지 명확하게 이해할 수 있도록 할 것이다.

실습 프로젝트: 성공하는 쇼핑몰

이번 강의는 우리가 직접 가상의 쇼핑몰을 만들고, 이 쇼핑몰의 데이터베이스를 설계하고 만들어가는 과정을 함께할 것이다.

초기 요구 사항

- 회원이 가입하고 자신의 정보를 관리할 수 있다.
- 판매자는 상품을 등록하고 관리할 수 있다.
- 회원은 상품을 주문할 수 있다.

이 간단한 요구 사항에서부터 시작해서 데이터베이스 설계의 모든 과정을 적용해볼 것이다.

강의 준비물

이번 강의를 수강하기 위해 필요한 준비물은 다음과 같다.

1. **MySQL 데이터베이스**: 이전 강의에서 설치했던 MySQL을 그대로 사용한다.
2. **데이터베이스 클라이언트**: DBeaver나 MySQL Workbench 등 손에 익은 GUI 툴을 사용하면 된다. 강의는 MySQL Workbench를 사용한다.
3. **백문이 불여일타**: 반드시 직접 실행해봐야 한다. 백번 보는 것 보다 한번 실행하는 것이 더 확실히 배울 수 있다.

이제 데이터베이스 설계의 세계로 떠나보자. 첫 번째 목적지는 '잘못된 설계가 부르는 재앙'을 직접 목격하는 현장이다.

> 실무 이야기

> 실무 개발에서 가장 중요한 것 하나만 선택하라고 한다면, 나는 주저 없이 데이터베이스 설계를 선택할 것이다.
>
> Java나 Python 같은 애플리케이션 코드는 상대적으로 수정하기 쉽다. 최신 프레임워크와 아키텍처 패턴 덕분에 기능 개선이나 코드 리팩토링이 과거보다 훨씬 유연해졌다. 하지만 데이터베이스 스키마, 즉 테이블의 구조는 한번 잘못 만들어지면 바로잡는 데 엄청난 비용과 시간이 소요된다. 데이터베이스는 모든 애플리케이션의 가장 깊은 곳에 있는 '뼈대'와 같기 때문이다.
>
> 건물의 기둥을 옮기려면 건물 전체를 들어내야 하는 것과 같은 이치다. 잘못된 설계는 결국 성능 저하, 데이터 불일치, 유지보수의 어려움이라는 '기술 부채'로 돌아온다. 그리고 이 부채는 시간이 지날수록 눈덩이처럼 불어나 결국 시스템 전체를 마비시키는 원인이 된다. 반면, 잘 된 설계는 어떤 비즈니스 요구 사항 변화에도 유연하게 대처할 수 있는 튼튼한 기반이 되어준다.
>
> 이 강의를 통해 여러분이 바로 그 '튼튼한 기반'을 다지는 능력을 갖추게 되기를 바란다.

잘못된 설계가 부르는 재앙

우리는 방금 데이터베이스 설계가 집을 짓기 전 설계도를 그리는 것과 같다고 이야기했다. 그렇다면 만약 설계도를 무시하고 집을 지으면 정확히 어떤 문제들이 발생할까? 데이터베이스 세계에서는 어떤 일이 벌어지는지, '나쁜 설계'가 실제로 어떤 재앙을 불러오는지 구체적인 사례를 통해 알아보자.

잘못된 설계가 부르는 재앙

이제 막 시작한 스타트업 '쇼핑몰'이 있다고 가정하자. 개발팀은 빠르게 서비스를 출시하기 위해 데이터베이스 설계에 많은 시간을 들이지 않고, 눈앞의 기능 구현에만 집중했다. 처음에는 모든 것이 순조로워 보였다.

무엇이 '나쁜 설계'인가?

개발팀은 고객과 주문 정보를 관리하기 위해 다음과 같은 `orders` 테이블 하나를 만들었다. 모든 정보를 한곳에 담아두면 관리하기 편할 것이라고 생각했다.

orders 테이블 구조

- `order_id`: 주문 번호
- `customer_id`: 고객 아이디

- `customer_name`: 고객 이름
- `customer_address`: 고객 주소
- `product_id`: 상품 번호
- `product_name`: 상품 이름
- `product_price`: 상품 가격
- `ordered_at`: 주문 날짜

그리고 이 테이블에는 다음과 같은 데이터가 쌓이기 시작했다.

order_id	customer_id	customer_name	customer_address	product_id	product_name	product_price
1001	user1	네이트	서울시 강남구	P001	좋은 키보드	50000
1002	user2	이철수	경기도 성남시	P002	편한 마우스	30000
1003	user1	네이트	서울시 강남구	P002	편한 마우스	30000
1004	user3	박영희	서울시 서초구	P003	고성능 모니터	200000
1005	user1	네이트	서울시 강남구	P003	고성능 모니터	200000

- 지면상 일부 컬럼은 생략한다.

겉보기에는 아무 문제가 없어 보인다. 하지만 이 설계는 이미 여러 개의 시한폭탄을 품고 있다. 지금부터 이 폭탄들이 어떻게 터지는지 하나씩 살펴보자.

나쁜 설계의 3대 문제점

나쁜 설계는 크게 세 가지 문제를 일으킨다. 바로 **데이터 무결성 훼손**, **성능 저하**, 그리고 **유지보수 비용 증가**다.

1. 데이터 무결성 훼손 (신뢰도 하락)

데이터 무결성이란 데이터가 항상 정확하고 일관된 상태를 유지하는 것을 의미한다. 나쁜 설계는 이 무결성을 아주 쉽게 깨뜨린다.

데이터 중복

위 테이블을 다시 보자. '네이트' 고객은 3번 주문했다. 그때마다 `customer_id`, `customer_name`, `customer_address`가 반복해서 저장된다. '편한 마우스' 상품 정보도 2번 주문될 때마다 `product_id`, `product_name`, `product_price`가 중복 저장되고 있다. 지금은 데이터가 몇 건 없어서 괜찮아 보이지만, 수백만

건의 주문이 쌓인다면 어떨까? 엄청난 양의 데이터가 불필요하게 중복되어 저장 공간을 낭비하게 된다. 이것은 사소한 시작에 불과하다. 진짜 문제는 '이상 현상(Anomaly)'에서 시작된다.

수정 이상 (Update Anomaly)

어느 날 '네이트' 고객이 이사를 가서 주소를 '서울시 강남구'에서 '제주시 애월읍'으로 변경해달라고 요청했다. 개발자는 `UPDATE` 쿼리를 실행해야 한다.

```
UPDATE orders
SET customer_address = '제주시 애월읍'
WHERE customer_id = 'user1';
```

이 쿼리를 실행하면 '네이트' 고객의 모든 주문 데이터의 주소가 한 번에 바뀔까? 그렇다. 지금 구조에서는 `WHERE customer_id = 'user1'` 조건으로 모든 데이터를 찾아서 변경할 수 있다.

하지만 만약 개발자가 실수로 다음과 같이 쿼리를 작성했다면 어떻게 될까?

```
UPDATE orders
SET customer_address = '제주시 애월읍'
WHERE order_id = 1005; -- 가장 최근 주문에만 주소를 변경
```

이 쿼리를 실행한 후의 테이블 상태를 보자.

[실행 결과]

order_id	customer_id	customer_name	customer_address	product_id	product_name	product_price
1001	user1	네이트	서울시 강남구	P001	좋은 키보드	50000
1002	user2	이철수	경기도 성남시	P002	편한 마우스	30000
1003	user1	네이트	서울시 강남구	P002	편한 마우스	30000
1004	user3	박영희	서울시 서초구	P003	고성능 모니터	200000
1005	user1	네이트	**제주시 애월읍**	P003	고성능 모니터	200000

끔찍한 일이 벌어졌다. '네이트'라는 동일한 고객의 주소가 주문에 따라 달라졌다. 어떤 주소가 진짜 주소인가? 데이터의 일관성이 깨져버렸다. 이제 이 데이터는 더 이상 신뢰할 수 없는 상태가 되었다. 이것이 바로 **수정 이상**이다. 하나의 정보를 바꾸기 위해 여러 데이터를 수정해야 하고, 그 과정에서 일부가 누락될 때 데이터 불일치가 발생하는 현상이다.

삽입 이상 (Insertion Anomaly)
'쇼핑몰'이 사업을 확장해서, 아직 주문은 하지 않았지만 마케팅 수신에 동의한 잠재 고객을 미리 등록하고 싶어졌다.
'션'이라는 신규 고객(`user4`)을 시스템에 추가해보자.

```
INSERT INTO orders (customer_id, customer_name, customer_address)
VALUES ('user4', '션', '부산시 해운대구');
```

이 SQL은 실행될까? `order_id`, `product_id`, `product_name`, `product_price`, `ordered_at` 같은 컬럼에 어떤 값을 넣어야 할까? 이 고객은 아직 주문한 적이 없으므로 주문 관련 정보는 없다. 만약 이 컬럼들이 `NOT NULL` 제약조건을 가지고 있다면, 데이터 삽입 자체가 불가능하다. 설령 `NULL`을 허용한다고 해도, 주문과 관련 없는 불필요한 `NULL` 데이터들이 테이블에 쌓이게 된다.

즉, 이 테이블 구조에서는 **주문을 하지 않으면 고객 정보를 등록할 수 없는 모순**이 발생한다. 이것이 바로 **삽입 이상**이다. 불필요한 정보(주문 정보)를 함께 넣어야만 원하는 데이터(고객 정보)를 저장할 수 있는 불합리한 상황이다.

삭제 이상 (Deletion Anomaly)
시간이 흘러 '이철수' 고객이 탈퇴를 요청했다. 법적인 보관 기간이 지나 그의 유일한 주문 기록(`order_id` = 1002)을 삭제해야 하는 상황이다.

```
DELETE FROM orders WHERE customer_id = 'user2';
```

이 쿼리를 실행하면 주문 정보가 깨끗하게 삭제된다. 하지만 동시에 무슨 일이 일어나는가? '이철수'라는 고객이 우리 쇼핑몰을 이용했다는 **고객 정보 자체가 시스템에서 완전히 사라져 버린다.** 나중에 데이터 분석을 위해 '과거에 어떤 고객들이 있었지?'를 알고 싶어도, 그의 주문 내역을 삭제하는 순간 고객 정보까지 함께 소멸되는 것이다.

이처럼 특정 정보를 삭제했을 뿐인데, 유지되어야 할 다른 중요한 정보까지 연쇄적으로 삭제되는 현상을 **삭제 이상**이라고 한다.
잘못된 설계는 데이터의 신뢰도를 근본부터 흔들어 놓는다.

2. 성능 저하 (속도 저하)

서비스가 성장하면서 `orders` 테이블에는 수억 건의 데이터가 쌓였다. 이제 이 거대한 테이블에서 데이터를 조회하는 속도가 눈에 띄게 느려지기 시작한다.

느려지는 조회 속도: '네이트' 고객의 주문 목록을 조회하는 간단한 쿼리를 생각해보자.

```
SELECT * FROM orders WHERE customer_id = 'user1';
```

이 쿼리는 매번 거대한 `orders` 테이블 전체를 뒤져야 한다. 고객 정보, 상품 정보, 주문 정보가 모두 한 테이블에 섞여 있으니 테이블이 '뚱뚱해지고' 길어진다. 디스크에서 읽어야 할 데이터 양이 많아지니 당연히 속도는 느려진다.

3. 유지보수 비용 증가 (확장성 저하)

비즈니스는 계속 변한다. 새로운 요구 사항이 생길 때마다 데이터베이스 구조도 함께 발전해야 한다. 하지만 잘못된 설계는 이 변화의 발목을 잡는다.

작은 변경의 큰 파급효과: 기획팀에서 '고객별 등급(BRONZE, SILVER, GOLD)을 관리하고 싶다'는 새로운 요구 사항을 제시했다. 이 간단한 '등급' 정보를 추가하려면 어떻게 해야 할까? `orders` 테이블에 `customer_grade`라는 컬럼을 추가해야 한다.

```
ALTER TABLE orders ADD COLUMN customer_grade VARCHAR(10);
```

그리고 '네이트' 고객의 등급을 'GOLD'로 올리려면, 그의 모든 주문 데이터를 일일이 찾아가며 등급을 수정해야 한다.

```
UPDATE orders SET customer_grade = 'GOLD' WHERE customer_id = 'user1';
```

앞서 살펴본 수정 이상 문제가 여기서도 동일하게 발생한다. 더 큰 문제는, 고객과 관련된 아주 작은 정보 하나를 추가했을 뿐인데, 수억 건의 주문 데이터가 담긴 거대한 `orders` 테이블의 구조를 변경해야 한다는 점이다. 이는 데이터베이스에 엄청난 부하를 주는 위험한 작업이 될 수 있다.

애플리케이션 로직의 복잡성 증가: 위에서 본 '이상 현상'들을 피하기 위해 개발자들은 애플리케이션 레벨에서 방어 로직

을 짜기 시작한다. 예를 들어, 고객 주소를 변경할 때는 `customer_id`로 모든 주문을 찾아서 주소를 업데이트하는 코드를 반드시 넣어야 하고, 신규 고객을 등록할 때는 주문 정보 없이 고객 정보만 임시 테이블에 넣었다가 첫 주문 시 `orders` 테이블로 옮기는 등의 복잡한 로직을 추가하게 된다. 데이터베이스가 제 역할을 못하니, 그 부담이 고스란히 애플리케이션 코드로 전가되어 시스템 전체가 비대하고 복잡해지는 결과를 낳는다.

극단적인 예시지만 이것이 바로 '나쁜 설계'가 불러오는 문제들이다. 데이터는 뒤죽박죽이 되고, 시스템은 느려지며, 작은 변화에도 전체가 휘청거린다. 우리는 이런 문제를 해결하기 위해 설계를 배운다.

설계의 3단계 - 개념, 논리, 물리

앞서 본 재앙과 같은 문제들을 체계적으로 해결하기 위해, 우리는 표준화된 설계 프로세스를 따라야 한다. 마치 건축가가 스케치에서 시작해 정교한 설계도를 그리고, 마지막으로 시공 계획을 세우는 것처럼 데이터베이스 설계도 3단계의 과정을 거친다.

이 3단계 로드맵은 우리가 가야 할 길을 알려주는 지도와 같다. 지금부터 각 단계가 무엇을 의미하고 어떤 역할을 하는지 알아보자.

설계의 3단계 - 목표

1단계: 개념적 설계 (Conceptual Design)

- **목표**: 비즈니스의 아이디어나 요구 사항을 이해하고, 현실 세계의 정보들을 컴퓨터 세상의 언어로 번역하기 위한 밑그림을 그리는 단계다.
- **핵심 질문**: "우리가 다루어야 할 데이터는 무엇이며(Entity), 그 데이터들은 서로 어떤 관계(Relationship)를 맺고 있는가?"
- **산출물**: **ERD (Entity-Relationship Diagram)**. 사람의 눈으로 가장 이해하기 쉬운 형태의 설계도다. `고객`은 `주문`을 할 수 있고, 하나의 `주문`에는 여러 개의 `상품`이 포함될 수 있다는 식의 관계를 그림으로 표현한다.
- **비유**: **건축가의 스케치**. 건물의 전체적인 모습과 각 공간의 배치를 간략하게 그리는 단계와 같다.

2단계: 논리적 설계 (Logical Design)

- **목표**: 개념적 설계에서 만든 밑그림을, 우리가 사용하려는 데이터베이스 기술인 '관계형 데이터베이스'의 원리에 맞게 구체적인 구조로 다듬는 단계다. 특정 RDBMS(MySQL, Oracle 등)에 종속되지 않는, 순수한 논리적인 데이터 구조를 만든다.
    - 논리적 설계의 핵심은 관계형 데이터베이스 이론에 맞게 설계하는 것이다. 다만 특정 관계형 데이터베이스

에 종속되게 만들면 안된다.

- **핵심 질문**: "개념 모델의 각 요소들을 어떤 테이블(Table) 구조로 표현할 것인가? 데이터의 중복을 막고 관계를 명확히 하기 위해 어떤 규칙(정규화)을 적용해야 하는가?"
- **산출물**: **정규화된 테이블 스키마**. '고객' 엔티티는 `고객` 테이블로, '상품' 엔티티는 `상품` 테이블로 만들고, 각 테이블이 어떤 컬럼(속성)들을 가질지, 기본 키(PK)와 외래 키(FK)는 무엇으로 할지 등을 명확하게 정의한다.
- **비유**: **건축 설계도 (평면도, 구조도)**. 스케치를 바탕으로 각 방의 정확한 크기, 창문의 위치, 구조물의 재질 등을 명시한 상세 설계도를 작성하는 단계와 같다.

3단계: 물리적 설계 (Physical Design)

- **목표**: 논리적 설계에서 만든 테이블 스키마를, 실제 우리가 사용할 특정 RDBMS(이 강의에서는 MySQL)의 특성에 맞게 최적화하여 구현하는 마지막 단계다.
- **핵심 질문**: "각 컬럼에 어떤 데이터 타입(VARCHAR, INT, DATETIME 등)을 할당해야 가장 효율적일까? 어떤 컬럼에 인덱스(Index)를 설정해야 조회 속도가 빨라질까?"
- **산출물**: **물리적인 테이블 정의서, SQL 스크립트 (`CREATE TABLE ...`)**. 이 스크립트를 실행하면 마침내 실제 데이터베이스에 테이블이 생성된다.
- **비유**: **시공 계획서**. 설계도를 현실에 구현하기 위해 어떤 철근을 사용할지, 어떤 순서로 공사를 진행할지 등 구체적인 실행 계획을 세우는 단계와 같다.

설계의 3단계 - 용어 정리

개념, 논리, 물리 각각의 설계 모델에 따라서 사용하는 용어가 다르다.

구분	개념 모델	논리 모델	물리 모델	엑셀 비유	쇼핑몰 예시
저장 구조	엔티티 (Entity)	릴레이션 (Relation)	테이블 (Table)	시트 (Sheet)	`회원 (user)`
세부 항목	속성 (Attribute)	속성 (Attribute)	열, 컬럼 (Column)	열 (Column)	회원의 이름, 주소
데이터 단위	인스턴스 (Instance)	튜플 (Tuple)	행 (Row)	행 (Row)	'네이트' 회원

각 모델링 단계마다 용어가 다른 이유는 각 단계가 가진 **고유한 목표와 바라보는 관점이 다르기 때문**이다. 데이터베이스를 설계하는 과정은 마치 집을 짓는 과정과 같다. 건축주와 이야기하는 단계, 설계도를 그리는 단계, 실제 시공하는 단계의 언어가 모두 다른 것과 같은 이치다.

1. 개념 모델: "어떤 집을 원하는가?" (비즈니스 관점)

- **목표**: 비즈니스의 요구 사항을 이해하고, 핵심 데이터가 무엇인지 정의하는 것. 개발자뿐만 아니라 기획자, 현업 담당자 등 **비전문가와 소통**하는 것이 주된 목적이다.
- **관점**: 현실 세계, 비즈니스 논리
- **사용 용어**: `엔티티(Entity)`, `속성(Attribute)`, `관계(Relationship)`

이 단계는 건축가가 건축주에게 "어떤 공간들이 필요한가? 침실은 3개, 화장실은 2개, 그리고 넓은 거실이 있었으면 좋겠다" 와 같이 원하는 바를 듣고 스케치하는 과정과 같다.

여기서 사용하는 `엔티티(사물)`, `속성(특징)` 같은 단어는 IT 전문 용어라기보다는 현실 세계의 대상을 표현하는 추상적이고 보편적인 용어다. '회원', '상품'이라는 중요한 **'대상(엔티티)'**이 있고, 각 대상은 '이름', '가격' 같은 **'특징(속성)'**을 가지며, 서로 '주문한다'는 **'관계'**를 맺는다고 표현해야 모두가 이해하기 쉽다. 만약 이 단계에서부터 `테이블`, `기본 키(PK)`, `외래 키(FK)` 같은 기술 용어를 사용한다면 비전문가와의 원활한 소통이 어려울 것이다.

결론: 개념 모델의 용어는 '현실 세계를 추상적으로 표현'하고 '원활하게 소통'하기 위해 존재한다.

2. 논리 모델: "어떻게 구조를 잡을 것인가?" (관계형 데이터베이스 구조 관점)

- **목표**: 개념 모델에서 파악한 요구 사항을 **관계형 데이터베이스 이론에 맞게** 체계적으로 구조화하는 것. 특정 데이터베이스 제품(MySQL, Oracle 등)에 종속되지 않는, 표준화된 설계도를 만드는 단계이다.
- **관점**: 관계형 데이터베이스 이론, 데이터의 논리적 구조
- **사용 용어**: `릴레이션(Relation)`, `속성`, `튜플(Tuple)`, `기본 키(PK)`, `외래 키(FK)`

이 단계는 건축가의 스케치를 바탕으로, 어떤 자재를 쓸지 정하기 전에 먼저 구조적으로 튼튼하고 효율적인 '건축 설계도(청사진)'를 그리는 과정과 같다. 기둥의 위치, 벽체의 연결, 각 방의 배치 등을 정하는 것이다.

논리 모델은 관계형 데이터베이스 이론을 기반으로 한다. 학술적으로는 `테이블` 대신에 `릴레이션`, `행` 대신에 `튜플` 같은 용어를 사용하지만, 실무에서는 이런 학술적인 용어를 거의 사용하지 않는다. 대신에 일반적으로 자주 사용하는 용어인 `테이블`, `컬럼`, `행`이라는 용어를 그대로 사용한다.

또한, 개념 모델에서는 그저 '관계'라고 표현했던 것을, 테이블 간의 연결을 구현하는 구체적인 기술인 `기본 키(PK)`와 `외래 키(FK)`로 명시한다. 특히 M:N 관계를 연결 테이블로 해소하는 등, 관계형 데이터베이스의 규칙에 맞게 데이터를 정규화하는 중요한 작업이 이 단계에서 이루어진다.

결론: 논리 모델의 용어는 '관계형 데이터베이스의 규칙에 맞게 데이터를 구조화'하기 위한 표준 언어이다.

> **릴레이션(Relation)**
> 릴레이션은 관계형 데이터베이스의 이론적 모델에서 '테이블(Table)'을 부르는 공식적인 이름이다. '테이블'이 실용적인 용어라면, '릴레이션'은 그 밑에 깔린 수학적이고 규칙적인 개념을 강조하는 용어이다.
> 릴레이션은 데이터가 저장되는 2차원 테이블 그 자체로 수학적 집합 개념을 뜻한다.
> 관계형 데이터베이스는 **'릴레이션(테이블)의 집합으로 구성된 데이터베이스'**를 의미하며, 이 릴레이션들을 이용해 데이터 간의 **관계(Relationship)**를 표현하는 것이다.
>
> **관계**형 데이터베이스라는 용어에서 사용하는 관계(Relation)라는 뜻은 테이블 간의 관계가 아니라 사실은 테이블(릴레이션) 그 자체를 뜻하는 것이다.

3. 물리 모델: "어떤 자재로 어떻게 시공할 것인가?" (물리적 구현 관점)

- **목표**: 논리 모델에서 만든 설계도를 **특정 데이터베이스 시스템(예: MySQL)에 실제로 구현**할 수 있도록 구체적인 명세를 정하는 것.
- **관점**: 실제 성능, 저장 공간, 특정 DBMS의 기술
- **사용 용어**: `테이블(Table)`, `컬럼(Column)`, `행(Row)`, `데이터 타입(VARCHAR, INT)`, `인덱스(Index)`

이 단계는 건축 설계도를 가지고, 'A사 시멘트', 'B사 창호' 등 실제 자재를 선택하고, 전선을 몇 밀리미터짜리로 설치할지 정하는 '시공 계획서'를 작성하는 것과 같다.

여기서는 논리 모델의 `테이블`과 `컬럼` 개념을 그대로 사용하지만, 훨씬 더 구체적인 정보가 추가된다. `name` 컬럼은 `VARCHAR(50)`으로, `price` 컬럼은 `INT`로 **데이터 타입을 지정**하고, 검색 성능을 높이기 위해 어떤 컬럼에 `인덱스`**를 걸지 결정**한다. 또한 `NOT NULL` 같은 제약 조건이나 `ENGINE=InnoDB` 같은 스토리지 엔진 옵션 등, 해당 DBMS가 제공하는 기능을 최대한 활용하여 최적의 성능과 안정성을 낼 수 있도록 설계한다.

결론: 물리 모델의 용어는 '특정 데이터베이스에 최적화된 방식으로 데이터를 구현'하기 위한 구체적인 언어이다.

이처럼 각 단계의 목표와 관점이 다르기 때문에, 그에 맞는 가장 효율적이고 정확한 용어를 사용하는 것이다. 이 과정을 통해 우리는 막연한 비즈니스 요구 사항을 체계적이고 성능 좋은 데이터베이스 시스템으로 완성해 나갈 수 있다.

> **실무 용어**
> 이론적으로는 용어들을 구분하지만, 실무에서는 이런 용어들을 엄격하게 구분해서 사용하지는 않는다. 예를 들어서 개념 모델을 설계할 때 엔티티라는 용어 대신에 테이블이라는 용어를 자주 사용하고, 속성이라는 용어 대신에 열(컬럼)이라는 용어도 자주 사용한다. 그리고 관계형 데이터베이스를 목표로 설계할 때는 개념 설계와 논리 설계를 함께 섞어서 진행하기도 한다. 따라서 이 부분은 엄격하게 구분하기 보다는 참고만 하자.

요약 및 다음 단계

지금까지 우리는 설계가 왜 중요한지를 느끼고, 앞으로 이 문제를 해결하기 위해 어떤 과정을 거쳐야 하는지(개념적-논리적-물리적 설계)에 대한 큰 그림을 확인했다.

기억해야 할 핵심은 이것이다. **좋은 설계는 '비용'이 아니라 '투자'다.** 초반에 들이는 설계 시간은 미래에 발생할 엄청난 재앙을 막고, 시스템의 수명을 늘리는 가장 현명한 투자다.

다음 시간부터는 이 로드맵의 첫 번째 단계인 **'개념적 설계'**를 본격적으로 시작할 것이다. 우리의 '쇼핑몰' 프로젝트 요구사항 속에서 핵심 데이터를 추출하고, 그것을 ERD라는 그림으로 그려내는 방법을 배워보자.

정리

데이터베이스 설계의 첫걸음

- 데이터베이스 설계는 집을 짓기 전 설계도를 그리는 것과 같이 시스템의 뼈대를 만드는 중요한 과정이다.
- 체계적인 설계 없이 데이터베이스를 구축하면 처음에는 빠를 수 있으나, 서비스가 복잡해질수록 데이터 중복, 성능 저하, 유지보수 어려움 등의 '기술 부채'가 쌓인다.
- 좋은 데이터베이스를 만들기 위해서는 개념적, 논리적, 물리적 설계의 3단계 과정을 거쳐야 한다.
- 이 과정은 데이터의 중복을 제거하고 무결성을 높이는 '정규화'를 포함한다.
- 잘 된 설계는 비즈니스 요구 사항 변화에 유연하게 대처할 수 있는 튼튼한 기반이 된다.

잘못된 설계가 부르는 재앙

- 잘못된 설계는 데이터 무결성 훼손, 성능 저하, 유지보수 비용 증가라는 세 가지 큰 문제를 일으킨다.
- **데이터 무결성 훼손**: 데이터 중복으로 인해 발생하며, 다음과 같은 이상 현상(Anomaly)을 유발한다.
  - **수정 이상**: 하나의 정보를 바꾸기 위해 여러 데이터를 수정해야 하고, 일부가 누락되면 데이터 불일치가 발생한다.
  - **삽입 이상**: 불필요한 정보 없이는 원하는 데이터를 저장할 수 없는 모순이 발생한다.
  - **삭제 이상**: 특정 정보를 삭제하면 유지되어야 할 다른 중요 정보까지 함께 사라진다.
- **성능 저하**: 불필요한 데이터까지 한 테이블에 저장하여 테이블이 비대해지면 조회 속도가 크게 느려진다.
- **유지보수 비용 증가**: 간단한 요구 사항을 추가할 때도 거대한 테이블 구조를 변경해야 하는 위험이 따르며, 데이터베이스의 문제를 해결하기 위해 애플리케이션 코드가 복잡해진다.

설계의 3단계 - 개념, 논리, 물리

- 데이터베이스 설계는 개념적, 논리적, 물리적 3단계의 표준화된 프로세스를 따른다.
- **1단계: 개념적 설계(Conceptual Design)**
  - 비즈니스 요구 사항을 이해하고 핵심 데이터(엔티티)와 그들 간의 관계를 파악하는 단계이다.
  - 주요 산출물은 ERD(Entity-Relationship Diagram)이다.
- **2단계: 논리적 설계(Logical Design)**
  - 개념 모델을 관계형 데이터베이스 이론에 맞게 테이블 구조로 구체화하는 단계이다.
  - 데이터 중복을 제거하는 정규화 과정을 거치고 기본 키(PK), 외래 키(FK) 등을 정의한다. 특정 데이터베이스 제품에 종속되지 않는다.
- **3단계: 물리적 설계(Physical Design)**
  - 논리 모델을 실제 사용할 특정 데이터베이스(예: MySQL)에 최적화하여 구현하는 단계이다.
  - 컬럼별 데이터 타입, 인덱스 등을 결정하고 `CREATE TABLE` 스크립트를 작성한다.

- 각 설계 단계는 바라보는 관점이 다르기 때문에 엔티티(개념) ➔ 릴레이션(논리) ➔ 테이블(물리)과 같이 사용하는 용어가 다르지만, 실무에서는 혼용하기도 한다.