11. 물리적 모델링

#0.강의/2.데이터베이스로드맵/3.설계1

물리적 모델링 개요

이제 데이터베이스 설계의 마지막 단계를 향해 가보자. 우리는 앞서 요구사항을 분석하고, 그 결과를 바탕으로 **개념적 모델링(ERD)**을 통해 데이터의 뼈대를 잡았다. 그리고 **논리적 모델링**을 통해 그 뼈대를 정규화하며 중복을 제거하고 데이터 무결성을 높이는 작업을 했다.

그렇다면 이제 이 설계도를 가지고 바로 데이터베이스를 구축하면 될까? 우리가 멋진 집을 설계했다고 가정하자. 설계도에는 방의 크기, 창문의 위치, 문의 개수 같은 논리적인 구조가 모두 담겨있다. 하지만 이 설계도만 가지고 실제로 집을 지을 수 있을까?

아니다. 실제 집을 지으려면 '벽은 어떤 재료로 만들 것인가?', '창문은 이중창으로 할까, 단일창으로 할까?', '전기 배선은 어떻게 깔 것인가?'와 같이 아주 구체적이고 현실적인 결정이 필요하다.

데이터베이스 설계도 마찬가지다. 논리적 모델링까지 마친 우리의 설계도는 아직 '어떤 데이터베이스(MySQL, Oracle, PostgreSQL 등)를 사용할지', '각 컬럼에 어떤 데이터 타입을 쓸지', '어떻게 하면 더 빠르게 데이터를 조회할 수 있을지'와 같은 현실적인 고민이 담겨있지 않다.

바로 이 단계가 **물리적 모델링(Physical Modeling)**이다. 논리적인 데이터 구조를 우리가 사용하기로 한 **MySQL**이라는 구체적인 데이터베이스 시스템에 최적화된 형태로 구현하는 과정인 것이다.

물리적 모델링이란?

물리적 모델링은 논리적 모델을 특정 데이터베이스 관리 시스템(DBMS)의 특성과 성능을 고려하여 구체적인 저장 구

조, 즉 물리적 스키마로 변환하는 과정이다.

여기서 가장 중요한 키워드는 **'성능'**과 **'특정 DBMS'**이다. 논리적 모델링이 데이터 구조의 정합성과 안정성에 초점을 맞췄다면, 물리적 모델링은 실제 시스템의 성능, 저장 공간 효율성, 유지보수 편의성 등을 극대화하는 데 목표를 둔다.

우리가 만든 쇼핑몰이 대성공을 거둬서 하루에 수백만 건의 주문이 들어온다고 상상해보자. 논리적으로 완벽한 모델이라 할지라도, 물리적 설계가 엉망이라면 주문 하나 처리하는 데 몇 초씩 걸릴 수도 있다. 그런 쇼핑몰은 아무도 사용하지 않을 것이다. 바로 이런 재앙을 막기 위해 우리는 물리적 모델링을 배우는 것이다.

논리적 모델링과 물리적 모델링 비교

논리적 모델링은 데이터의 논리적 무결성과 관계를 중시하며, 물리적 모델링은 이를 바탕으로 실제 구현 세부 사항을 추가한다.

측면	논리적 모델링 (Logical Modeling)	물리적 모델링 (Physical Modeling)
DBMS	**DBMS에 독립적** (특정 제품에 종속되지 않음)	**DBMS에 종속적** (MySQL, Oracle 등 특정 제품을 선택해서, 특성 반영)
목적	데이터의 논리적 구조 정의 (테이블, 키, 관계 등) 비즈니스 규칙을 관계형 모델로 변환	논리적 모델을 실제 DBMS에 최적화된 형태로 구현. 성능, 저장, 접근 효율성을 고려
주요 작업	- 엔티티를 테이블(릴레이션)로 변환 - 속성 정의 (추상적 데이터 타입, 기본 키, 외래 키) - 관계(1:1, 1:N, N:N)를 정의 - 정규화 수행 (1NF ~ BCNF) - 제약 조건 정의 (NOT NULL, UNIQUE 등)	- 물리적 테이블(Table), 컬럼(Column), 데이터 타입(Data Type) 정의 - 테이블 스페이스나 파일 그룹 할당 - 인덱스(Index) 생성 (클러스터드/논클러스터드, B-tree 등) - 파티셔닝(Partitioning) 설계 (범위, 해시 등으로 데이터 분할) - 스토리지 엔진 선택 (InnoDB등 DBMS 별) - 뷰, 트리거, 저장 프로시저의 물리적 구현 - 보안 설정 (접근 제어, 암호화) - 성능 튜닝 (쿼리 최적화, 캐싱 전략)
고려 요소	- DBMS 독립적 (추상적) - 데이터 무결성 - 관계와 중복 최소화	- DBMS 종속적 (특정 벤더의 기능 활용) - 하드웨어 자원 (디스크, 메모리) - 쿼리 패턴과 워크로드 분석 - 확장성(Scalability)과 백업/복구

도구/출력물	- ERD (논리적 데이터 모델) - ER 다이어그램에서 관계형 스키마로 변환. - DDL 스크립트 초안 (CREATE TABLE 등). - 용어 사전 (개념적 모델과 연결)	- 최종 DDL 스크립트 (인덱스, 파티션 포함) - 테이블 정의서 - 물리적 다이어그램 - 성능 테스트 스크립트
순서 및 의존성	개념적 모델링 후 수행. 물리적 모델링의 기반이 됨	논리적 모델링 후 수행. 실제 배포 전 테스트 필요
예시	고객 엔티티 → Customer 테이블 (ID PK, Name String, Address String). 관계: Orders 테이블과 1:N	Customer 테이블에 index 생성. 속성 타입 정의, 데이터 파티션: 연도별로 분할. InnoDB 엔진 사용

물리적 모델링 프로세스

물리적 모델링은 다음과 같은 단계로 진행된다.

1. **테이블과 컬럼 변환**: 논리적 모델의 한글 이름을 영문 이름으로 변환한다. 이때 테이블명, 컬럼명을 정하는 규칙(Naming Convention)을 정하고 적용한다. (용어 사전 참고)
2. **데이터 타입 정의**: 각 컬럼에 가장 적합한 데이터 타입(예: `VARCHAR`, `INT`, `DATETIME`)을 선택한다. 이는 저장 공간과 성능에 직접적인 영향을 미친다.
3. **제약 조건 설정**: 기본 키, 외래 키, `NOT NULL` 등 데이터의 무결성을 보장하기 위한 제약 조건을 구체적으로 설정한다.
4. **인덱스 설계**: 데이터 조회 성능을 극대화하기 위해 어떤 컬럼에 인덱스를 생성할지 결정한다.
5. **역정규화 및 성능 튜닝**: 필요에 따라 정규화 원칙을 위배하여 테이블을 통합하거나 중복 데이터를 추가하는 '역정규화'를 수행하여 성능을 개선한다. (최후의 수단)
6. **파티셔닝, 샤딩등 기타 기법 적용**: 대용량 테이블의 경우, 특정 기준으로 데이터를 분할하여 저장하는 파티셔닝 같은 고급 기법을 고려한다.
7. **뷰, 저장 프로시저, 함수, 트리거 생성**: 완성된 테이블 구조 위에서 필요한 뷰, 프로시저, 트리거 등 추가적인 데이터베이스 객체를 생성한다.

이제 물리적 모델링이 무엇인지, 왜 필요한지 감이 잡혔을 것이다. 이제부터 이 프로세스를 하나씩 따라가면서 우리의 쇼핑몰 데이터베이스를 실제로 구축할 수 있는 수준으로 구체화해 보자.

물리적 모델링에서 다루는 구현 내용은 실전 데이터베이스 입문편과 기본편에서 이미 학습한 내용들이다.

데이터 타입, 테이블 생성, 제약 조건, 인덱스, 뷰, 저장 프로시저, 함수, 트리거 등 물리적 모델링의 실제 구현 방법들은 실전 데이터베이스 입문편, 기본편에서 이미 다룬 내용들이다. 따라서 이미 학습한 내용은 제외하거나 간단하게 줄여서 언급하겠다.

인덱스

인덱스는 실전 데이터베이스 기본편에서 다음과 같은 주제로 자세히 설명했다. 자세한 내용은 기본편을 참고하자.

- 인덱스가 필요한 이유
- 트리 자료 구조
- 인덱스 생성, 조회, 삭제
- 인덱스와 비교 연산(동등, 범위, LIKE)
- 인덱스와 정렬
- 옵티마이저와 인덱스 선택
- 커버링 인덱스
- 복합 인덱스
- 인덱스 실행 계획 확인
- 인덱스 설계 가이드라인
- 인덱스의 단점과 주의사항

뷰, 저장 프로시저, 함수, 트리거

뷰, 저장 프로시저, 함수, 트리거는 실전 데이터베이스 기본편에서 다루었다. 자세한 내용은 기본편을 참고하자.

파티셔닝과 샤딩

파티셔닝이나 샤딩과 같은 대용량 테이블에 맞는 고급 기법은 실전 데이터베이스 - 성능 최적화와 고급 기능 편에서 다룬다.

테이블과 컬럼 변환 규칙1 - 기본 규칙

논리적 모델링에서 정의한 내용을 이제 실제 물리적인 테이블과 컬럼으로 변환할 차례다. 이 과정에서 아무렇게나 이름을 짓는 것이 아니라, 프로젝트 전체의 일관성과 가독성을 높이고 미래의 유지보수를 용이하게 만들기 위해 명확한 '규칙'을 따라야 한다. 이를 **명명 규칙(Naming Convention)**이라고 한다. 잘 정립된 명명 규칙은 그 자체로 훌륭한 문서가 된다.

> **절대적인 규칙은 없다.**
> 개발을 할 때 절대적인 규칙은 없다. 프로젝트의 종류와 규모에 따라서 규칙은 얼마든지 달라질 수 있다.
> 같은 규칙이라고 해도 프로젝트 종류와 규모에 따라서 잘 맞는 경우도 있고, 잘 맞지 않는 경우도 있다.
> 중요한 것은 우리 프로젝트에 이런 규칙을 적용했을 때 어떤 장점이 있고, 어떤 단점이 있는지 진지하게 고민하고 결정하는 것이다. 그리고 그 결정을 일관성 있게 적용하는 것이다.

일반적인 명명 규칙 (Naming Convention)

실무적으로 널리 사용되는 일반적인 규칙은 다음과 같다.

1. **영어 사용**: 데이터베이스 객체(테이블, 컬럼 등)의 이름은 영어를 사용하는 것이 국제적인 표준이다. 한글 이름을 사용할 경우, 개발 환경이나 라이브러리에서 호환성 문제가 발생할 수 있다.

2. **소문자 스네이크 케이스 (`snake_case`)**: 테이블과 컬럼 이름은 모두 소문자로 작성하고, 단어가 여러 개 조합될 때는 언더스코어(`_`)로 연결하는 '스네이크 케이스'를 사용하는 것이 일반적이다. 예를 들어, `order_item`이나 `created_at`과 같다. 대문자를 섞어 쓰는 것은 운영체제에 따라 테이블 이름을 다르게 인식하는 문제를 일으킬 수 있으므로 권장하지 않는다.

3. **명확하고 서술적인 이름**: 이름만 보고도 그 역할과 의미를 명확히 알 수 있도록 짓는다. 의미 없는 축약은 피하는 것이 좋다. 예를 들어 `member_registration_datetime`는 `mem_reg_dt`보다 훨씬 이해하기 쉽다.
    - 물론 팀에서 사용하는 축약어에 `mem`, `reg`, `dt`를 등록해 두었다면 `mem_reg_dt`를 사용하는 것이 좋다.

4. **일관성 있는 접두사와 접미사**: 이름에 일관성을 부여하면 가독성이 크게 향상된다.
    - 예를 들어 모든 테이블의 기본 키(PK) 컬럼은 `테이블명_id` 형식(예: `member_id`, `product_id`)을 따르는 것이 좋다.

5. **예약어 사용 금지**: `ORDER`, `GROUP`, `SELECT`, `CREATE` 등 SQL에서 이미 특별한 의미를 가지고 있는 예약어(Reserved Words)는 테이블이나 컬럼 이름으로 사용해서는 안 된다.

- 주문 테이블의 경우 이런 이유로 예외적으로 `order` 대신에 `orders`를 사용한다.

컬럼 이름 규칙

참고하면 좋은 컬럼 이름 규칙은 다음과 같다.

1. 기본 키(PK)는 '테이블명_id'로 명명한다

가장 중요하고 기본적인 규칙 중 하나다. 모든 테이블의 기본 키를 그냥 `id`라고 이름 짓는 경우를 생각해 보자. 당장은 편해 보일 수 있지만, 여러 테이블을 조인(JOIN)할 때 문제가 발생한다.

예를 들어, `member` 테이블과 `orders` 테이블에 모두 `id`라는 이름의 PK가 있다고 가정해 보자.

```
SELECT
    id, -- 이 id가 member의 id인가, orders의 id인가?
    member_name,
    ordered_at
FROM
    member
JOIN
    orders ON member.id = orders.member_id;
```

위 SQL을 실행하면 데이터베이스는 `id` 컬럼이 어느 테이블에 속한 것인지 알 수 없어 `Column 'id' in field list is ambiguous` (필드 목록의 'id' 컬럼이 모호하다) 라는 오류를 발생시킨다. 물론 `member.id`처럼 테이블 이름 또는 별칭을 명시하여 해결할 수 있지만, 모든 쿼리에서 이렇게 별칭을 붙이는 것은 매우 번거롭다.

이제 우리가 설계한 것처럼 `member_id`, `order_id`로 명명했을 때의 장점을 보자.

```
SELECT
    m.member_id,
    o.order_id,
    m.member_name,
    o.ordered_at
FROM
    member m
JOIN
    orders o ON m.member_id = o.member_id;
```

컬럼 이름만으로도 어떤 테이블의 ID인지 명확하게 구분된다. 이는 쿼리의 가독성을 극적으로 높여주고, 잠재적인 실수를 줄여준다. **컬럼 이름 그 자체로 데이터의 출처와 의미를 설명한다.**

2. 외래 키(FK)는 참조하는 테이블의 PK 이름을 그대로 사용한다

이 규칙은 1번 규칙과 자연스럽게 이어진다. 외래 키는 다른 테이블과의 관계를 나타내는 중요한 열쇠이므로, 어떤 테이블을 참조하는지 이름만 보고도 알 수 있어야 한다.

우리 설계에서 `orders` 테이블의 `member_id` 컬럼을 보자.

- `orders` 테이블의 `member_id`는 `member` 테이블의 `member_id`를 참조하는 외래 키다.
- 이름이 같기 때문에 개발자는 `orders` 테이블의 `member_id`가 `member` 테이블과 관련이 있다는 것을 즉시 유추할 수 있다.

만약 `orders` 테이블에서 FK로 사용하는 회원 ID를 `customer_id`와 같이 다른 이름으로 지었다면,두 테이블의 관계를 파악하기 위해 매번 테이블의 구조를 확인해야 하는 번거로움이 생긴다.

3. 날짜/시간 컬럼은 접미사로 용도를 명확히 한다

데이터가 언제 생성되었는지, 언제 수정되었는지, 또는 특정 이벤트가 언제 발생했는지를 기록하는 날짜/시간 컬럼은 매우 흔하다. 이런 컬럼들에는 일관된 접미사를 붙여주는 것이 좋다.

- `_at`: 특정 작업이 발생한 정확한 시점(날짜와 시간)을 기록할 때 사용한다. 'at a specific time'의 의미를 가진다.
  - 주로 특정 사건(이벤트)가 발생한 시간을 기록할 때 사용한다. 주문 시각, 취소 시각, 로그인 시각등
  - 예: `created_at`, `updated_at`, `deleted_at`, `logined_at`, `ordered_at`, `paid_at`
- `_datetime`, `dt`: 일시, 날짜와 시간 정보가 중요할 때 사용한다. 너무 길기 때문에 `dt`로 많이 줄여서 사용한다.
  - 예: `start_dt`, `end_dt`, `reservation_dt`
  - 예약일시, 이벤트 시작/종료 시각이나 예약 시각처럼 특정 기간이나 시점을 명시할 때 유용하다.
- `_date`: 날짜 정보에만 사용한다. 시간 정보가 필요 없거나 의미가 없는 경우에 적합하다.
  - 예: `birth_date`, `join_date`, `release_date` (회원의 생년월일이나 상품 출시일 등이 좋은 예다.)
- `_time`: 시간 정보에만 사용한다. 날짜와 관계없이 특정 시간 정보가 필요할 때 사용한다.
  - 예: `opening_time`, `closing_time`, `notification_time` (상점의 영업 시작 및 종료 시간이나, 매일 반복되는 알림 시간 설정 등에 사용할 수 있다.)

용어 사전에 추가 - 날짜/시간

이런 규칙을 확정했다면 용어 사전에 추가해서 관리하는 것이 좋다.

분류	명칭	전체 영문명	축약어	설명	관련 시스템 요소
날짜/시간	이벤트가 발생한 일시(날짜+시간)	`at`	`at`	과거 특정 이벤트가 발생한 **정확한 시점(날짜와 시간)**을 기록, 주문, 결제 가입, 취소등 `~에`, `~시점에`라는 의미를 가진 영어 전치사 `at` 사용	`ordered_at` `paid_at` `created_at` `updated_at`
	이벤트 발생이 아닌 일시(날짜+시간)	`datetime`	`dt`	미래의 '예정' 또는 사용자가 지정한 변경 가능한 시간 (예약, 희망 배송일 등)	예약 시간 `reservation_dt` 희망 배송일 `desired_delivery_dt`
	날짜	`date`		시간을 제외한 날짜	
	시간	`time`		날짜를 제외한 시간	

4. 불리언(Boolean) 타입 컬럼은 'is_' 또는 'has_' 접두사를 사용한다

참(True) 또는 거짓(False) 상태를 나타내는 불리언 컬럼은 그 의미를 명확히 하기 위해 `is_`나 `has_`와 같은 접두사를 붙이는 것이 좋다. 이는 쿼리의 `WHERE` 절을 읽을 때 마치 자연스러운 문장처럼 만들어준다.

예를 들어, '탈퇴한 회원'을 저장하는 컬럼 이름이 그냥 `withdrawal`이라고 가정해 보자.
`WHERE withdrawal = TRUE`
이것이 탈퇴했다는 의미인지, 탈퇴가 가능하다는 의미인지 즉시 파악하기 어렵다.

하지만 `is_withdrawal` 이라고 명명하면 의미가 명확해진다.
`WHERE is_withdrawal = TRUE` (탈퇴했는가? = 그렇다)

다른 좋은 예시는 다음과 같다.

- `is_active` : 현재 활성 상태인가?
- `has_coupon` : 쿠폰을 가지고 있는가?
- `is_shipped` : 배송이 완료되었는가?

5. 컬럼은 단수 명사 사용
컬럼은 테이블의 한 행(Row)이 가지는 단일 속성을 나타내므로, 단수 명사를 사용해야 한다.

- **예시**: `product_name` (O), `product_names` (X).
- **이유**: 테이블 이름은 집합을 의미하므로 규칙을 정하기에 따라 복수형을 쓰기도 하지만, 컬럼은 단일 값을 저장하므로 반드시 단수형을 사용해야 한다.

6. 의미 있는 축약어
팀 내에서 모두가 동의하고 그 의미를 명확히 알 수 있는 축약어가 아니라면, 완전한 단어를 사용하는 것이 좋다.
팀 내에서 모두 동의하는 축약어는 `qty(quantity)` , `id(identifier)` 는 단 2가지라고 가정하자.

- **좋은 예시**: `quantity` 대신 `qty` , `identifier` 대신 `id` .
- **나쁜 예시**: `stock` 을 `stk` 로, `member_registration_date` 를 `mem_reg_dt` 로 줄이는 것은 가독성을 해친다.
- **이유**: 명확성은 간결성보다 우선이다. 몇 글자를 아끼기 위해 미래의 개발자가 의미를 추측하게 만드는 것은 장기적으로 더 큰 비용을 발생시킨다.

7. 데이터의 성격을 명확히 표현

- **단위나 종류 명시**: 만약 우리 쇼핑몰이 여러 국가의 통화를 지원한다면, `price` 대신 `price_krw` , `price_usd` 처럼 통화 단위를 명시해 주는 것이 좋다. 무게를 다룬다면 `weight_kg` , `weight_g` 처럼 단위를 포함하는 것이 혼동을 막는다.
- **문맥 제공**: `product` 테이블의 상품명 컬럼은 `product_name` 대신에 `name` 도 간결하고 괜찮아 보인다. `product` 라는 테이블의 문맥 안에서는 `name` 이 상품명을 의미한다는 것을 쉽게 알 수 있기 때문이다. 하지만 여러 테이블을 `JOIN` 한 결과에서는 `name` 보다는 `product_name` 이 더 명확할 수 있다. 예를 들어 `member` 테이블에도 `name` 이 있다면, 별칭을 통한 둘의 구분이 필요하기 때문이다. 따라서 문맥을 활용하는 방식을 기본으로 하되, 여러 테이블에서 자주 사용되는 용어라면 구분을 고려하자.

일관성의 중요성

위에서 제시한 규칙들은 널리 사용되는 좋은 관례이지만, 절대적인 법칙은 아니다. 가장 중요한 원칙은 '일관성'이다. 하나의 프로젝트에서는 모든 팀원이 동의한 하나의 명명 규칙을 정하고, 데이터베이스 전체에 걸쳐 일관되게 적용해야 한다. 일관성만 있다면 설령 조금 특이한 규칙이라도 괜찮다. 일관성이 깨지는 순간, 데이터베이스는 이해하기 어렵고 유지보수하기 힘든 골칫덩어리가 될 수 있다.

테이블과 컬럼 변환 규칙2 - 축약어와 단수 복수

축약어 사용의 역사와 함정

과거의 데이터베이스 시스템은 이름 길이에 8자, 16자 같은 엄격한 제약이 있었다. 이 때문에 축약어 사용이 불가피했지만, 현대의 데이터베이스 시스템에서는 이런 제약이 거의 사라졌다. 또한, 과거에는 지금처럼 강력한 자동 완성(Auto-completion) 기능을 갖춘 IDE나 SQL 클라이언트가 드물었다.

그럼에도 불구하고 타이핑을 줄여준다는 단기적인 편리함 때문에 축약어를 사용하고 싶은 유혹에 빠지기 쉽다. 하지만 의미가 불분명한 축약어는 시간이 지나면서 그 의미를 파악하기 어렵게 만들어 결국 유지보수 비용을 증가시키는 '기술 부채(Technical Debt)'로 쌓이게 된다.

좋은 축약어의 조건

물론 모든 축약어가 나쁜 것은 아니다. 좋은 축약어는 오히려 가독성을 높이고, 널리 사용되어 의미 파악에 전혀 문제가 없다. 좋은 축약어는 다음 조건을 만족해야 한다.

- **보편성 (Universality)**: `id`(identifier), `avg`(average), `max`(maximum), `min`(minimum), `qty`(quantity)처럼 누가 봐도 그 의미를 알 수 있는, 업계에서 널리 통용되는 축약어여야 한다.
- **비모호성 (Unambiguity)**: 반드시 하나의 의미로만 해석되어야 한다. 예를 들어 `desc`라는 컬럼 이름은 'description'(설명)과 SQL 예약어인 'descending'(내림차순)으로 모두 해석될 수 있어 혼란을 야기한다. `auth` 역시 'author'(저자), 'authentication'(인증), 'authorization'(권한 부여) 등 다양한 의미를 가질 수 있어 피해야 한다.
- **일관성 및 문서화 (Consistency & Documentation)**: 만약 특정 비즈니스 도메인에서만 통용되는 축약어를 사용해야 한다면, 반드시 프로젝트의 '데이터 사전'이나 '용어집'에 그 의미를 명확히 기록해야 한다. 그리고 프로젝트의 모든 데이터베이스에서 일관되게 사용해야 한다.
    - 금융권 - `pnl`(profit and loss)
    - 쇼핑몰 - `sku`(Stock Keeping Unit)

나쁜 축약어가 초래하는 장기적 비용

나쁜 축약어는 당장의 편리함보다 훨씬 큰 비용을 장기적으로 발생시킨다.

- **유지보수 비용 증가**: 데이터베이스를 처음 접하는 개발자는 축약어의 의미를 파악하기 위해 추가적인 시간을 소모해야 한다. `mem_reg_dt`가 `member_registration_date`의 줄임말임을 추측하는 과정은 불필요한 인지 부하를 유발하고 생산성을 떨어뜨린다. 물론 `mem`, `reg`, `dt`를 팀에서 합의한 축약어라면 사용할 수 있다.
- **버그 발생 가능성 증가**: 축약어를 오해하면 잘못된 쿼리를 작성하여 심각한 버그로 이어질 수 있다. `auth_status`가 'authorization'(권한 부여) 상태인지, 'authentication'(인증) 상태인지 불분명하다면 보안 관련 로직에 치명적인 결함을 만들 수 있다.

핵심 원칙은 이것이다. **"의심스러우면, 축약하지 말고 전부 써라"** 잘 지은 컬럼 이름은 그 자체로 훌륭한 문서의 역할을 한다.

균형을 찾는 가이드라인

물론 실무에서는 명확성과 편의성 사이의 균형점을 찾는 지혜가 필요하다. 컬럼 이름이 `last_successful_login_attempt_from_mobile_device_timestamp` 처럼 지나치게 길어지면 오히려 쿼리 작성과 가독성을 해칠 수 있다.

핵심 원칙은 **'모호함(Ambiguity)을 제거하는 것'**이다. 이 원칙을 지키면서 이름을 적절히 줄일 수 있는 몇 가지 실무 가이드를 제시하겠다.

1. 테이블 이름으로 컨텍스트를 파악할 수 있다면 과감히 생략해라.
가장 흔하고 효과적인 방법이다. 컬럼은 독립적으로 존재하지 않고 항상 테이블이라는 소속(컨텍스트)이 있다.

- **나쁜 예시**: `member` 테이블 안에 `member_name`, `member_email`, `member_address` 라는 컬럼을 만드는 것.
- **좋은 예시**: `member` 테이블 안에서는 그냥 `name`, `email`, `address` 라고 지으면 된다.

`SELECT name, email FROM member` 쿼리를 보면, 이 `name`과 `email`이 `member`의 정보라는 것을 누구나 알 수 있다. `member_` 라는 접두사는 불필요하게 이름만 길게 만드는 군더더기다.

단 이때 너무 많은 테이블에서 같은 이름을 사용한다면 접두사 사용을 고려하자.
예)

- `member`, `product` 등등 수 많은 테이블에서 `name`이라는 이름을 사용한다. ➜ `member_name`,

`product_name` 등으로 접두사 고려

- `email`: `member` 테이블을 포함한 몇 개의 테이블만 사용한다. ➜ `email` 그대로 사용
- `status`: `orders`, `product`, `pay`, `delivery` 등 수 많은 테이블에서 상태를 나타내기 위해 사용한다. ➜ `order_status`, `product_status` 등 접두사 고려

핵심은 기본적으로 테이블 명이라는 문맥을 활용하되, 너무 많은 테이블에서 사용해서 조인 시 자주 충돌이 일어나고, 구분이 어렵다면 접두사 사용을 고려하자.

2. 보편적인 약어는 적극적으로 활용해라.
앞서 '좋은 축약어의 조건'에서 이야기했듯이, 모두가 아는 약어는 쓰는 것이 좋다.

- `identification` ➜ `id`
- `quantity` ➜ `qty`
- `average` ➜ `avg`
- `number` ➜ `no` (예: `post_no`)
- `datetime` ➜ `dt` (단, `_at`과 같은 접미사 규칙과 충돌하지 않도록 팀 내 합의가 필요하다)

예를 들어 `product_average_rating_point`는 `product_avg_rating` 정도로 충분히 의미를 전달하면서 길이를 줄일 수 있다.

3. 이름이 너무 길다면, 모델링이 잘못된 것은 아닌지 의심해라.
때로는 긴 컬럼 이름 자체가 **설계의 문제점**을 알려주는 신호(Code Smell)일 수 있다.

예를 들어, `member` 테이블에 `primary_shipping_address_street_and_postal_code` 라는 컬럼이 있다고 가정해보자. 이름이 매우 길고 복잡하다.

이것은 '주소'라는 개념을 별도의 컬럼이나 테이블로 분리하지 않았기 때문에 발생하는 문제다. 이 컬럼을 만드는 대신, `address` 테이블을 새로 만들고 `street`, `postal_code`, `is_primary` 같은 컬럼으로 분리하는 것이 올바른 정규화 설계이다. 즉, 긴 이름의 문제를 이름 자체를 줄여서 해결하는 것이 아니라 **구조적인 개선**으로 해결해야 하는 경우다.

위에서 제시한 방법들을 통해 **'모호하지 않은 선에서 가장 간결한 이름'**을 찾는 연습을 해야 한다. 중요한 것은 타이핑 몇 번을 줄이는 편의성이 아니라, 6개월 뒤에 다른 동료(또는 미래의 나 자신)가 이 컬럼을 봤을 때 단 1초의 망설임도 없이 의미를 파악하게 만드는 것이다.

테이블 이름: 단수(Singular) vs 복수(Plural)

테이블 이름을 단수로 할지, 복수로 할지에 대한 논쟁은 오래전부터 있었다.

- **복수형 주장 (`members`, `products`, `orders`)**: 테이블은 여러 개의 데이터(레코드)가 모여있는 '집합'이므로, 그 의미를 살려 복수형을 써야 한다는 주장이다. `SELECT * FROM members` 처럼 SQL 쿼리가 좀 더 자연스럽게 읽히는 장점이 있다.
- **단수형 주장 (`member`, `product`, `order`)**: 테이블은 해당 엔티티(개체)의 '설계도' 또는 '틀'을 정의하는 것이므로, 엔티티의 이름 그대로 단수형을 써야 한다는 주장이다. `member` 테이블의 한 행(row)이 한 명의 회원을 의미하므로, 이쪽이 개념적으로 더 명확하다고 볼 수 있다. 특히 ORM(Object-Relational Mapping) 기술과 함께 사용할 때 클래스 이름과 매핑하기 용이하다.

결론적으로 어떤 것을 선택하든 정답은 없다. **하지만 프로젝트 내에서는 반드시 하나를 선택해서 일관성을 지켜야 한다.** 이 강의에서는 개념적 모델의 엔티티 이름을 그대로 따라가는 **단수형**을 기준으로 설명하겠다. 우리가 앞서 만든 DDL에서도 `member`, `product`, `orders` (`order`는 예약어라 `orders`로 사용)와 같이 단수형을 기반으로 이름을 지었다.

한국에서 단수형을 주로 사용하는 이유

한국 개발 환경에서는 언어적 특성과 개발의 편의성이라는 실용적인 이유로 단수형 테이블 이름을 압도적으로 선호한다. 그 이유는 다음과 같다.

- **언어적 특성:** 한국어는 명사의 단수와 복수를 엄격하게 구분하지 않는다. '사용자'라는 단어는 한 명의 사용자를 지칭할 수도 있고, 여러 명의 사용자를 의미할 수도 있다. 따라서 영어처럼 'User'와 'Users'를 구분해야 할 필요성을 크게 느끼지 못하며, 더 간결하고 기본적인 형태인 단수형을 자연스럽게 선호한다.
- **영문법의 부담 감소:** 영어가 모국어가 아닌 입장에서 불규칙한 복수형(예: `Person` → `People`, `Analysis` → `Analyses`)을 매번 정확히 사용하는 것은 번거롭고 실수의 여지가 있다. 단수형으로 통일하면 이러한 문법적 고민 없이 단순하게 일관된 규칙을 유지할 수 있어 개발 효율성이 높아진다.

> **데이터베이스 입문편과 기본편에서 복수형을 사용한 이유**
>
> - `order`와 같은 일부 단어는 데이터베이스에서 사용하는 **예약어**와 충돌한다. 예약어와 충돌하는 단어를 테이블 이름으로 사용하려면 매번 백틱(`)을 추가해야 하는 번거로움이 있다.
> - 예제 코드를 **일관성 있게** 유지하기 위해 모든 테이블 이름을 복수형으로 통일했다. 만약 `order`만 `orders`로 사용하고 다른 테이블은 단수형을 사용했다면, 복수형을 사용한 이유에 대해 미리 설명해야 하는 복잡함이 생긴다.

용어 사전 활용

개념적, 논리적 모델링 단계에서는 기획자나 다른 팀원들과의 원활한 소통을 위해 '회원', '주문', '상품'과 같은 한글 용어를 사용하는 경우가 많다. 이는 비즈니스 로직을 이해하는 데 매우 효과적이다.

하지만 물리적 모델링으로 넘어오면서 이 한글 이름들은 앞서 설명한 규칙에 따라 반드시 **영문으로 변환**되어야 한다. 이때 개발자마다 '주문 항목'을 `order_detail`로 할지, `order_item`으로 할지 다르게 생각할 수 있다. 이러한 불일치를 막아주는 것이 바로 **용어 사전**이다.

설계 초기에 용어를 표준화해두면, 기획자, 개발자, DBA 등 프로젝트에 참여하는 모든 사람이 동일한 용어를 사용하게 되어 소통의 오류를 줄일 수 있다. 또한, 나중에 물리 모델링을 할 때 어떤 영어 단어를 써야 할지 고민하는 시간을 줄여주고, 이름의 비일관성으로 인해 발생하는 버그를 사전에 방지하는 효과도 있다.

우리는 앞서 개념적 모델링 단계부터 용어 사전을 정의했다. 용어 사전 덕분에 편리하게 일관성 있게 영문으로 된 물리적 이름을 변환할 수 있다.

데이터 타입1 - 문자, 숫자, PK 타입

논리적 모델의 '속성'을 물리적 모델의 '컬럼'으로 바꿀 때, 가장 먼저 해야 할 일은 각 컬럼에 어떤 종류의 데이터가 들어갈지 정의하는 것이다. 이것이 바로 **데이터 타입**을 지정하는 작업이다.

왜 데이터 타입을 신중하게 골라야 할까?

문제 상황: 우리 쇼핑몰에 '회원' 테이블이 있다고 해보자. 회원의 나이(age)를 저장해야 한다. 이때 어떤 개발자가 '나이는 숫자니까... 그냥 넉넉하게 큰 숫자를 담을 수 있는 타입으로 하자!' 라며 `BIGINT` 타입을 선택했다고 가정하자. `BIGINT`는 약 922경까지 저장할 수 있는 아주 큰 숫자 타입이다. 사람의 나이가 200살을 넘기는 경우는 거의 없는데, 이렇게 큰 타입을 쓰는 것이 과연 효율적일까?

답은 '아니오'다. 이는 마치 라면 하나를 끓이려고 거대한 솥을 사용하는 것과 같다. 필요 이상의 공간을 차지하고, 데이터를 처리할 때도 더 많은 리소스를 사용하게 되어 결국 성능 저하로 이어진다.

이처럼 데이터 타입을 잘못 선택하면 다음과 같은 문제가 발생한다.

- **저장 공간 낭비**: 필요보다 큰 데이터 타입을 사용하면 디스크 공간이 낭비된다. 데이터가 수백만, 수천만 건이 되면 이 낭비는 무시할 수 없는 수준이 된다.

- **성능 저하**: 데이터베이스는 디스크에서 메모리로 데이터를 읽어와 처리한다. 데이터 타입이 크면 한 번에 읽어올 수 있는 데이터의 양이 줄어든다. 이는 더 많은 I/O(입출력)를 유발하고, 결국 쿼리 속도를 느리게 만든다.
- **데이터 무결성 훼손**: 데이터의 성격에 맞지 않는 타입을 사용하면 잘못된 데이터가 입력될 수 있다. 예를 들어, 날짜를 문자열(`VARCHAR`)로 저장하면 '2025-08-21' 뿐만 아니라 '내일', '어제' 같은 엉뚱한 값도 들어갈 수 있다.

따라서 우리는 데이터의 특성과 범위를 정확히 파악하고, 그에 맞는 **최적의 데이터 타입**을 선택해야 한다. 이제 MySQL에서 자주 사용하는 주요 데이터 타입들을 하나씩 알아보자.

문자열 타입

문자열을 저장하는 타입은 크게 가변 길이와 고정 길이로 나뉜다.

- `VARCHAR(M)` : **가변 길이** 문자열. `M`은 저장할 수 있는 **최대 길이**를 의미한다. 실제 저장되는 데이터의 길이에 따라 저장 공간의 크기가 달라진다. 예를 들어 `VARCHAR(100)`에 'hello' (5글자)를 저장하면, 실제로는 5글자에 해당하는 공간 + 길이 정보(1~2바이트)만 사용한다. (최대 바이트 길이가 255 이하이면 1바이트, 그 초과이면 2바이트를 사용한다.)
    - **장점**: 공간 효율이 좋다.
    - **단점**: 길이가 변하기 때문에 데이터 수정 시 추가적인 작업이 필요할 수 있다.
    - **용도**: 이름, 제목, 주소 등 대부분의 문자열 데이터에 사용된다.
- `CHAR(M)` : **고정 길이** 문자열. `M`은 **고정된 길이**를 의미한다. `CHAR(10)`에 'hello' (5글자)를 저장하면, 나머지 5글자는 공백으로 채워져 무조건 10글자의 공간을 차지한다.
    - **장점**: 길이가 고정되어 있어 데이터 처리 속도가 `VARCHAR`보다 약간 빠를 수 있다.
    - **단점**: 공간 낭비가 심하다.
    - **용도**: 주민등록번호, 전화번호, 성별('M'/'F')처럼 길이가 항상 고정된 데이터에 사용할 수 있다.
- `TEXT` : 매우 긴 텍스트를 저장할 때 사용한다. 게시물의 본문, 상품의 상세 설명 등에 사용된다. (e.g., `TEXT`는 약 6만 5 바이트, `MEDIUMTEXT`는 약 1600만 바이트, `LONGTEXT`는 약 42억 바이트)

실무 가이드

- **"일단 `VARCHAR`를 써라."** 대부분의 경우 `VARCHAR`가 가장 합리적인 선택이다. 길이가 고정된 데이터라도 그 길이가 바뀔 가능성이 있다면 `VARCHAR`를 쓰는 것이 더 안전하다.
- `VARCHAR`의 길이는 얼마나 잡아야 할까? 너무 길게 잡으면 메모리 사용량이 늘어날 수 있고, 너무 짧게 잡으면 데이터가 잘리는 문제가 생긴다. 저장될 데이터의 평균적인 길이와 최대 길이를 고려하여 합리적으로 설정해야 한

다. 예를 들어, 사용자 이름은 50자, 이메일 주소는 100자 정도로 잡는 것이 일반적이다.

예제: 회원 정보를 저장하는 테이블을 만들어보자.

```
DROP TABLE IF EXISTS member_sample;

CREATE TABLE member_sample (
    member_id BIGINT PRIMARY KEY,
    email VARCHAR(100),    -- 이메일 주소는 길이가 다양하다.
    name VARCHAR(50),      -- 이름도 길이가 다양하다.
    gender CHAR(1),        -- 성별은 'M' 또는 'F'로 길이가 1로 고정된다.
    introduction TEXT      -- 자기소개는 매우 길어질 수 있다.
);

INSERT INTO member_sample (member_id, email, name, gender, introduction)
VALUES (1, 'test@example.com', '션', 'M', '안녕하세요. 데이터베이스를 배우는 션입니다.');
```

[심화] VARCHAR 길이는 왜 메모리 사용량에 영향을 줄까?
VARCHAR는 가변 길이인데 왜 길이를 크게 잡으면 메모리 사용량이 늘어날까?
디스크 저장 공간과 메모리 사용은 다른 이야기이기 때문이다.

결론부터 말하면, MySQL이 쿼리를 처리하기 위해 **메모리에 임시 공간을 할당할 때** `VARCHAR`에 설정된 **최대 길이**를 기준으로 삼는 경우가 많다.

예를 들어, 우리가 `ORDER BY`나 `GROUP BY` 같은 정렬이나 그룹화 작업을 수행한다고 생각해보자. MySQL은 효율적인 작업을 위해 디스크에 있는 데이터를 메모리로 가져와서 중간 결과물을 저장할 임시 테이블(temporary table)을 만들 수 있다.

이때, 메모리에 만들어지는 임시 테이블의 컬럼 크기는 원본 테이블의 `VARCHAR` 최대 길이를 따라간다.

- `name VARCHAR(50)` 컬럼을 정렬한다면: MySQL은 메모리에 한 사람의 이름을 저장하기 위해 **최대 50글자**를 담을 수 있는 공간을 할당한다. 실제 이름이 'Kim' (3글자)이더라도, 정렬 중 자리가 바뀔 수 있으므로 넉넉하게 최대 크기로 잡는 것이다.
- `name VARCHAR(4000)` 컬럼을 정렬한다면: MySQL은 같은 원리로 메모리에 **최대 4000글자**를 담을 수 있는 공간을 할당한다.

이제 수백만 건의 회원 데이터를 정렬한다고 상상해보자. `VARCHAR(50)`일 때와 `VARCHAR(4000)`일 때, 정렬 작업을 위해 필요한 총 메모리의 양은 어마어마한 차이가 난다. 만약 할당해야 할 메모리가 너무 커서 서버의 물리적인 메모리(RAM) 용량을 초과하면, MySQL은 디스크를 임시 공간으로 사용하기 시작한다. 디스크 I/O는 메모리 I/O보다 수천 배에서 수만 배 느리기 때문에, 이는 곧바로 쿼리 성능의 급격한 저하로 이어진다.

따라서 `VARCHAR`의 길이를 설정할 때는 "혹시 모르니 그냥 엄청 크게 잡자"가 아니라, "이 데이터가 가질 수 있는 합리적인 최대 길이가 얼마일까?"를 신중하게 고민해야 한다. 이것이 바로 디스크 공간 효율뿐만 아니라, 시스템 전체의 성능을 생각하는 좋은 데이터베이스 설계자의 자세이다.

숫자 타입

숫자 데이터는 정수형과 소수형으로 나뉜다.

정수 타입

정수를 저장할 때 사용하며, 저장할 수 있는 값의 범위에 따라 여러 타입으로 나뉜다.

타입	저장 공간 (Bytes)	최소값	최대값	용도 예시
`TINYINT`	1	-128	127	나이, 상태 코드
`SMALLINT`	2	-32,768	32,767	작은 범위의 개수
`MEDIUMINT`	3	-8,388,608	8,388,607	
`INT`	4	-2,147,483,648	2,147,483,647	일반적인 ID, 조회수, 재고 수량
`BIGINT`	8	약 -922경	약 922경	매우 큰 ID (주문 번호 등)

실무 가이드

- `TINYINT`는 상태 값(예: 0=대기, 1=처리중, 2=완료)이나 한정된 범위의 숫자에 사용하면 공간을 매우 효율적으로 쓸 수 있다.
- 일반적인 게시물 ID, 회원 ID 등은 `INT`로 충분한 경우가 많다. 하지만 양수만 저장하는 경우 `UNSIGNED` 옵션을 사용하면 저장 범위를 2배로 늘릴 수 있다. 예를 들어 `INT UNSIGNED`는 0부터 약 42억까지 저장할 수 있어, 음수가 필요 없는 ID 값에 사용하기 좋다.
- 앞으로 서비스가 엄청나게 커질 것을 대비해 무조건 `BIGINT`를 사용하는 경우가 있는데, 이는 신중해야 한다. 트

래픽이 많지 않은 관리자 페이지의 ID 등은 `INT`로도 충분하다. 하지만 사용자의 주문 ID처럼 데이터가 폭발적으로 증가할 것이 확실하다면 처음부터 `BIGINT`를 고려하는 것이 좋다. 나중에 `INT`에서 `BIGINT`로 바꾸는 작업은 매우 고통스럽기 때문이다.

소수 타입

소수점이 있는 숫자를 저장할 때 사용한다.

- `DECIMAL(M, D)`: 고정 소수점 타입. 금융 계산처럼 **정확한 소수점 계산이 필요할 때 반드시 사용**해야 한다. `M`은 총 자릿수, `D`는 소수점 이하 자릿수를 의미한다. 예를 들어 `DECIMAL(10, 2)`는 정수 부분 8자리, 소수 부분 2자리까지 총 10자리의 숫자를 정확하게 저장할 수 있다.
- `DOUBLE` **또는** `FLOAT`: 부동 소수점 타입. 과학 계산이나 근사치가 허용되는 빠른 계산에 사용된다. 하지만 소수점 계산 시 미세한 오차가 발생할 수 있어, 돈과 관련된 계산에는 절대 사용하면 안 된다.

예제: 상품 가격과 평점을 저장하는 테이블을 만들어보자.

```
DROP TABLE IF EXISTS product_sample;

CREATE TABLE product_sample (
    product_id BIGINT PRIMARY KEY,
    name VARCHAR(100),
    price DECIMAL(10, 2), -- 99999999.99 까지 저장 가능
    rating DOUBLE         -- 평점은 근사치로 저장해도 무방
);

INSERT INTO product_sample (product_id, name, price, rating)
VALUES (1, '프리미엄 키보드', 159000.00, 4.8);

INSERT INTO product_sample (product_id, name, price, rating)
VALUES (2, '고급 마우스', 89500.50, 4.9);
```

- `price`에 달러와 같은 소수점을 사용하는 통화라면 `DECIMAL`을 사용해서 정확도를 보장해야 한다.
- `rating`은 약간의 오차가 있어도 괜찮으므로 `DOUBLE`을 사용했다.

PK 타입 선정: INT vs BIGINT

기본 키(PK)를 자동 증가(Auto Increment)하는 숫자로 정했다면, 이제 어떤 숫자 타입을 쓸지 결정해야 한다. 현실적으로 선택지는 `INT`와 `BIGINT` 둘 중 하나로 좁혀진다.

과거에는 `INT`가 PK의 표준처럼 사용되었다. `INT`에 `UNSIGNED` 옵션을 적용하면 0부터 약 42억까지의 숫자를 저장할 수 있다. 42억이라는 숫자는 어지간한 서비스에서는 전부 소진하기 어려운 매우 큰 숫자이므로, 대부분의 경우에 충분했다.

하지만 현대의 웹 서비스, 특히 우리가 만들 쇼핑몰처럼 성공을 목표로 하는 서비스는 데이터 증가 속도를 예측하기 어렵다. 회원이 수천만 명이 되고, 한 회원이 여러 개의 주문을 생성하며, 각 주문마다 여러 개의 주문 상품 데이터가 쌓인다고 생각해보자. 핵심적인 데이터인 `orders`나 `order_item` 테이블은 수십억 건을 넘어설 가능성이 충분하다.

`INT` vs `BIGINT` 성능과 저장 공간

그렇다면 왜 고민하는가? 처음부터 가장 큰 `BIGINT`를 사용하면 모든 문제가 해결되지 않을까?

맞는 말이면서도 틀린 말이다. `BIGINT`는 `INT`에 비해 두 배의 저장 공간(8 bytes vs 4 bytes)을 사용한다. 이것이 단순히 디스크 공간만 더 차지하는 문제에서 그치지 않는다. 데이터베이스에서 성능에 가장 큰 영향을 미치는 것은 **I/O(디스크 입출력)**다.

데이터베이스는 인덱스라는 자료구조를 통해 데이터를 빠르게 찾는다. 이때 인덱스 정보도 결국 디스크에 저장되어 있고, 필요할 때 메모리로 읽어와야 한다. PK는 가장 중요한 인덱스다. 데이터 타입의 크기가 작을수록, 한 번에 메모리에 읽어올 수 있는 인덱스 데이터의 양이 많아진다. 즉, 더 적은 I/O로 원하는 데이터를 찾을 가능성이 커진다는 의미다. 따라서 `INT`를 사용하는 것이 `BIGINT`를 사용하는 것보다 **이론적으로 더 빠르고 효율적**이다.

BIGINT PK를 사용하는 이유

지금까지 이야기한 내용을 합리적으로 생각해보면 다음과 같은 가이드라인이 떠오를 것이다.

1. **회원, 상품, 주문, 결제 등 서비스의 핵심 데이터**: 사용자가 계속해서 만들어내는 데이터, 즉 앞으로 얼마나 늘어날지 예측하기 어려운 테이블의 PK는 고민하지 말고 `BIGINT`를 사용한다. 미래의 위험을 미리 방지하는 것이 현명한 선택이다. 우리 쇼핑몰 예제의 `member_id`, `product_id`, `order_id`가 모두 `BIGINT`인 이유다. (국내 서비스라면 회원의 경우에는 INT를 고려할 수 있다.)
2. **카테고리, 상태 코드 등 내부적으로 사용하는 데이터**: 데이터의 최대 개수가 수백, 수천 개 수준으로 명확하게 예측 가능한 테이블(예: 상품 카테고리, 주문 상태 코드 테이블)은 `INT`를 사용하는 것이 효율적이다.

그럼에도 불구하고, **결론부터 말하면 실무에서는 핵심 테이블의 PK로 `BIGINT` 사용을 권장한다.**

지금부터 그 이유를 자세히 알아보자.

`INT`와 `BIGINT`의 저장 공간 차이 분석

`BIGINT`가 `INT`보다 2배의 공간을 사용하는 것은 사실이다. 하지만 이것이 실제로 얼마나 큰 차이를 만드는지 표로 확인해보자.

데이터 건수	INT 저장 공간 (4 bytes)	BIGINT 저장 공간 (8 bytes)	차이
1 건	4 Bytes	8 Bytes	4 Bytes
10 건	40 Bytes	80 Bytes	40 Bytes
100 건	400 Bytes	800 Bytes	400 Bytes
1,000 건	4 KB	8 KB	4 KB
10,000 건	39 KB	78 KB	39 KB
100,000 건	390 KB	781 KB	391 KB
1,000,000 건	3.8 MB	7.6 MB	3.8 MB
10,000,000 건	**38.1 MB**	**76.3 MB**	**38.1 MB**
100,000,000 건	**381.5 MB**	**762.9 MB**	**381.5 MB**
1,000,000,000 건	3.7 GB	7.4 GB	3.7 GB

- 쇼핑몰 판매자가 1000명이라면 `INT`와 `BIGINT` 차이는 겨우 `4 KB` 이다.
- 카테고리가 1만건이라면 `INT`와 `BIGINT` 선택의 차이는 겨우 `39 KB` 차이다.
- 상품 데이터가 100만건 이라면 `INT`와 `BIGINT` 선택의 용량 차이는 약 `3.8 MB`에 불과하다. 이는 최신 스마트폰의 사진 한 장보다 작은 용량이다.
- 참고로 데이터베이스 내부의 실제 부가 비용은 더 들 수 있다.

1억 건 정도 되어야 약 400MB로 어느정도 의미있는 수준의 차이가 나오기 시작한다. 이 정도 차이는 현대의 스토리지 기술 수준에서는 충분히 감당할 수 있는 수준이다. **게다가 데이터가 1억 건에 도달했다는 것 자체가 이미 서비스가 엄청나게 성장했다는 증거이며, `INT`를 사용했다면 이 시점에서는 최대 저장 용량 초과 위험 때문에 `BIGINT`로의 변경을 심각하게 고려해야 할 것이다.** 즉, `BIGINT`를 사용함으로써 잃는 저장 공간의 손해는 매우 미미한 반면, 얻게 되는 **'규칙의 단순함', '미래 확장성', '관리의 용이성'**이라는 이점은 매우 크다.

미래 확장성

`INT`를 사용해서 얻는 약간의 성능상 이점보다, 나중에 `INT`의 저장 범위를 초과했을 때 발생하는 **재앙적인 비용**이 훨씬 더 크다. 서비스가 한창 성장하는 와중에 PK 타입을 `INT`에서 `BIGINT`로 변경하는 작업(`Migration`)은 상상 이

상으로 고통스럽다. 서비스 전체를 중단해야 할 수도 있고, 데이터 정합성이 깨질 위험도 크다.

실무에서는 또 다른 관점, 바로 **'일관성'**의 가치를 매우 높게 평가한다. 데이터가 적을 것이라 예상되는 테이블에도 `BIGINT`를 적용하는 것이 장기적으로 더 나은 선택일 수 있다.

일관성이 주는 강력함
프로젝트의 모든 테이블 PK를 `BIGINT`로 통일하면 다음과 같은 장점을 얻을 수 있다.

1. **고민의 제거와 개발 생산성 향상**: 개발자는 새로운 테이블을 설계할 때마다 '이 테이블은 데이터가 많이 쌓일까? `INT`로 충분할까?'를 고민할 필요가 없다. **'PK는 `BIGINT`'** 라는 단순하고 명확한 규칙은 개발자의 인지 부하를 줄여주고, 더 중요한 비즈니스 로직에 집중하게 만든다.

2. **예측 실패의 위험 방지**: 지금은 데이터가 몇 개 없을 것 같은 '카테고리' 테이블이, 서비스가 고도화되면서 사용자가 직접 카테고리를 생성하고 태그처럼 사용하는 기능으로 확장될 수도 있다. 초기의 예측이 틀렸을 때 `INT`에서 `BIGINT`로 변경하는 비용은 매우 크다. 일관된 `BIGINT` 정책은 이러한 예측 실패의 위험을 원천적으로 차단하는 '보험'과 같다.

3. **외래키 관리의 단순함**: `orders` 테이블의 `member_id`는 `member` 테이블의 `member_id`를 참조하는 외래키(FK)다. FK는 참조하는 PK와 반드시 데이터 타입이 같아야 한다. 만약 어떤 PK는 `INT`이고 다른 PK는 `BIGINT`라면, FK를 설정할 때마다 참조하는 테이블의 PK 타입을 확인해야 하는 번거로움이 생긴다. 이는 실수로 이어질 가능성을 높인다. 모든 PK를 `BIGINT`로 통일하면, 모든 FK 또한 `BIGINT`로 만들면 되므로 실수를 줄일 수 있다.

따라서 많은 최근 실무 현장에서는 성능이나 저장 공간의 미미한 차이보다는, 장기적인 안정성과 유지보수 편의성을 위해 모든 테이블의 PK를 `BIGINT`로 통일하는 전략을 선호한다.

데이터 타입2 - 날짜와 시간 타입

날짜 및 시간 타입

날짜와 시간 정보를 저장하는 데 사용되는 타입은 매우 중요하다. 문자열로 저장하면 날짜 계산이나 비교가 매우 어렵기 때문이다.

- `DATE`: 날짜만 저장한다. 'YYYY-MM-DD' 형식. (예: '2025-08-21')

- `DATETIME` : 날짜와 시간을 함께 저장한다. 'YYYY-MM-DD HH:MI:SS' 형식. (예: '2025-08-21 11:30:00')
- `TIME` : 시간만 저장한다. 'HH:MI:SS' 형식.

실무 가이드

- 회원 가입일, 주문일, 게시물 작성일 등 대부분의 경우 `DATETIME` 을 사용한다.
- 생년월일처럼 시간 정보가 필요 없는 경우에는 `DATE` 를 사용한다.
- 특별한 이유가 없다면 `TIMESTAMP` 대신에 `DATETIME` 을 사용하는 것이 더 직관적이고 안전하다.
  - **데이터베이스 입문편에서 설명**
  - TIMESTAMP는 시간대 변환/범위(1970~2038) 제약이 있어 비즈니스 시각 기록은 `DATETIME` 이 일반적으로 안전.
- **"생성일과 수정일은 기본이다."** 실무에서는 대부분의 테이블에 `created_at` (생성일시)과 `updated_at` (수정일시) 컬럼을 추가한다. 이 데이터는 나중에 문제 추적이나 데이터 분석에 매우 유용하게 사용된다.

생성일과 수정일 자동화

`created_at` 과 `updated_at` 은 매우 중요하지만, 데이터를 삽입하거나 수정할 때마다 개발자가 직접 `NOW()` 함수를 사용해서 시간을 넣어주는 것은 번거롭고 실수를 유발하기 쉽다. 만약 깜빡하고 값을 넣지 않으면 데이터가 누락될 것이다.

데이터베이스는 이런 반복적인 작업을 자동화할 수 있는 매우 강력하고 편리한 기능을 제공한다.

- `DEFAULT CURRENT_TIMESTAMP` : 컬럼에 기본값을 현재 시간으로 설정한다. 데이터가 처음 삽입될 때, 해당 컬럼에 값을 명시하지 않으면 데이터베이스가 자동으로 현재 시간을 기록해준다.
- `ON UPDATE CURRENT_TIMESTAMP` : 해당 로우(row)의 데이터가 수정될 때마다, 자동으로 현재 시간으로 값을 갱신해준다.

이 두 가지 옵션을 사용하면 `created_at` 과 `updated_at` 을 개발자가 전혀 신경 쓰지 않아도 데이터베이스가 알아서 관리해준다.

예제: 생성일과 수정일이 자동 관리되는 게시판 테이블

게시판 테이블을 예시로 DDL을 작성해보자.

```
DROP TABLE IF EXISTS board_sample;

CREATE TABLE board_sample (
    board_id    BIGINT          NOT NULL AUTO_INCREMENT,
    title       VARCHAR(255)    NOT NULL,
```

```
    content     TEXT            NULL,
    created_at  DATETIME        NOT NULL DEFAULT CURRENT_TIMESTAMP,
    updated_at  DATETIME        NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
    PRIMARY KEY (board_id)
);
```

- `created_at`: `DEFAULT CURRENT_TIMESTAMP`를 적용해서, 데이터가 생성될 때의 시간이 자동으로 기록되도록 했다.
- `updated_at`: `DEFAULT CURRENT_TIMESTAMP`와 `ON UPDATE CURRENT_TIMESTAMP`를 모두 적용했다. 이렇게 하면 데이터가 처음 생성될 때는 `created_at`과 동일한 시간이 기록되고, 이후에 해당 데이터가 수정될 때마다 수정된 시점의 시간으로 자동으로 갱신된다.

자동화 기능 확인하기

이제 실제로 데이터베이스가 어떻게 자동으로 시간을 기록하는지 확인해보자.

1. 데이터 삽입

`created_at`과 `updated_at` 컬럼을 제외하고 `INSERT` 문을 실행한다.

```
INSERT INTO board_sample (title, content) VALUES ('첫 번째 게시글', '게시글 내용입니
다.');
```

2. 삽입 결과 확인

데이터를 조회해서 `created_at`과 `updated_at`에 값이 잘 들어갔는지 확인한다.

```
SELECT board_id, title, created_at, updated_at FROM board_sample;
```

[실행 결과]

board_id	title	created_at	updated_at
1	첫 번째 게시글	2025-09-06 11:35:05	2025-09-06 11:35:05

결과를 보면, `INSERT` 문에 시간을 전혀 명시하지 않았는데도 `created_at`과 `updated_at`에 정확한 생성 시각이 기록된 것을 확인할 수 있다.

3. 데이터 수정

이제 방금 삽입한 게시글의 내용을 수정해보자.

```
UPDATE board_sample
SET content = '게시글 내용이 수정되었습니다.'
WHERE board_id = 1;
```

4. 수정 결과 확인

다시 데이터를 조회해서 `updated_at`이 변경되었는지 확인한다.

```
SELECT board_id, title, created_at, updated_at FROM board_sample;
```

[실행 결과]

board_id	title	created_at	updated_at
1	첫 번째 게시글	2025-09-06 11:35:05	2025-09-06 11:36:00

결과를 보자. `created_at`은 처음 생성된 시간 그대로 유지되고, `updated_at` 컬럼만 데이터가 수정된 시간으로 깔끔하게 변경되었다.

이렇게 데이터베이스의 자동화 기능을 활용하면 데이터가 언제 변경되었는지 쉽게 추적할 수 있다. 그리고 이제 개발자는 비즈니스 로직에 더 집중할 수 있게 된다. 실무에서는 거의 모든 테이블에 이 방식을 적용한다고 생각하면 된다.

비즈니스 날짜 vs 자동 생성 날짜

우리 테이블의 `member` 테이블에는 회원이 가입한 시점을 기록하는 `created_at`이 있다. `orders` 테이블에는 `ordered_at`과 `created_at`이 둘 다 존재한다. 이 둘의 차이는 실무에서 매우 중요하다.

- `ordered_at`: 회원이 **'주문'이라는 비즈니스 행위를 한 시점**이다.

- `created_at`: 이 주문 데이터가 우리 데이터베이스 **테이블에 '생성'된 시점**이다.

대부분의 경우 두 시간은 거의 동일하겠지만, 데이터 마이그레이션이나 지연 처리 등으로 인해 달라질 수 있다. 이렇게 목적에 따라 컬럼 이름을 명확히 구분하면 데이터의 의미를 혼동 없이 정확하게 파악할 수 있다.

> **실무 팁**
> `created_at`은 데이터가 처음 데이터베이스에 저장된 물리적인 시간을 기록하는 용도로, `ordered_at`와 같은 컬럼은 주문 접수, 결제 완료 등 비즈니스 이벤트가 발생한 논리적인 시간을 기록하는 용도로 구분해서 사용하면 시스템을 더 정교하게 관리할 수 있다.

용어 사전 추가 - created at, updated at

이런 규칙을 확정했다면 용어 사전에 추가해서 관리하는 것이 좋다.

분류	명칭	전체 영문명	축약어	설명	관련 시스템 요소
행위/수식어	생성	`create`		테이블에 데이터가 만들어진 시점을 나타낼 때 사용. (예: `created_at`)	`created_at`
	수정	`update`		테이블에 데이터가 변경된 시점을 나타낼 때 사용. (예: `updated_at`)	`updated_at`

기타 타입

- `BOOLEAN` **(또는** `BOOL`**)**: 내부적으로는 `TINYINT(1)`로 처리된다. `TRUE`는 1, `FALSE`는 0으로 저장된다. 상품의 전시 여부, 회원의 탈퇴 여부 등 참/거짓 상태를 나타낼 때 유용하다. (0은 거짓, 0이 아니면 참으로 판별)
- `ENUM`: 미리 정해진 몇 개의 문자열 값 중 하나만 저장할 수 있는 타입이다. 예를 들어 회원의 등급을 'BRONZE', 'SILVER', 'GOLD' 중 하나로만 제한하고 싶을 때 `ENUM('BRONZE', 'SILVER', 'GOLD')`와 같이 정의할 수 있다. 실무에서는 잘 사용하지 않는다.
- `JSON`: JSON 형식의 데이터를 그대로 저장할 수 있는 타입이다. 스키마가 유동적인 데이터를 저장할 때 유용하지만, 관계형 데이터베이스의 장점을 제대로 활용하기 어려워 남용해서는 안 된다.

역정규화

논리적 모델링에서 데이터의 중복을 제거하고 정합성을 높이기 위한 '정규화'에 대해 깊이 있게 학습했다. 정규화는 관계형 데이터베이스 설계의 가장 기본적이고 중요한 원칙이다. 그렇다면, 정규화만 잘 따르면 항상 최고의 데이터베이스 설계가 되는 것일까?

안타깝게도 현실은 그렇게 간단하지 않다. 이번에는 정규화의 원칙을 의도적으로 위배하는 기술인 '역정규화(Denormalization)'에 대해 알아보자.

역정규화가 필요한 이유: 성능

정규화의 가장 큰 단점은 테이블이 잘게 분리된다는 점이다. 예를 들어, 사용자의 주문 내역과 각 주문에 포함된 상품명 리스트를 조회하려면 어떻게 해야 할까?

```
SELECT
    o.order_id,
    o.ordered_at,
    m.member_name,
    p.product_name,
    oi.order_quantity,
    oi.order_price
FROM orders o
JOIN member m ON o.member_id = m.member_id
JOIN order_item oi ON o.order_id = oi.order_id
JOIN product p ON oi.product_id = p.product_id
WHERE m.login_id = 'user123';
```

이처럼 간단한 조회 작업에도 `member`, `orders`, `order_item`, `product` 무려 4개의 테이블을 `JOIN` 해야 한다. 데이터가 적을 때는 문제가 없지만, 수백만 건의 주문 데이터가 쌓이고 수천 명의 사용자가 동시에 접속하는 쇼핑몰이라면 어떨까? 잦은 `JOIN` 연산은 데이터베이스에 부하를 주어 시스템 전체의 성능을 저하시키는 주범이 될 수 있다.

바로 이 **조회 성능** 문제를 해결하기 위해 등장한 것이 역정규화다.

> **역정규화(Denormalization)**
> 데이터의 조회 성능을 향상시키기 위해, 의도적으로 데이터 모델의 정규화 원칙을 위반하여 데이터의 중복

> 을 허용하는 프로세스다.

정규화와의 트레이드오프(Trade-off) 관계

역정규화란, 데이터베이스의 성능 향상을 위해 **의도적으로 정규화 원칙을 위배하고 데이터의 중복을 허용하는 프로세스**를 말한다. 주된 목표는 쿼리 실행 시 발생하는 `JOIN`의 횟수를 줄여서 조회 성능을 극대화하는 것이다.

정규화가 데이터를 '분해'하는 과정이라면, 역정규화는 필요한 데이터를 다시 '통합'하고 '중복'시키는 과정에 가깝다.

역정규화는 **데이터의 일관성과 정합성을 일부 희생**하고, **조회 속도를 얻는** 일종의 트레이드오프 관계에 있다. 따라서 반드시 신중하게 접근해야 한다.

역정규화를 이해하기 위해서는 정규화와의 트레이드오프 관계를 명확히 알아야 한다.

구분	정규화 (Normalization)	역정규화 (Denormalization)
목표	데이터의 **일관성** 및 **무결성** 확보	데이터 조회 **성능** 향상
장점	데이터 중복 최소화, 쓰기(C/U/D) 성능 유리	읽기(SELECT) 성능 향상 (`JOIN` 감소)
단점	읽기 성능 저하 가능성 (`JOIN` 증가)	데이터 **중복 증가**, 쓰기 성능 불리, 데이터 불일치 위험

역정규화는 **'읽기 속도를 위해 쓰기 속도와 데이터 일관성을 일부 희생하는'** 전략적인 선택이다. 따라서 아무 때나 적용해서는 안 된다. 반드시 정규화된 모델링을 기본으로 하고, 성능 테스트를 통해 명확한 병목 지점이 확인되었을 때 최후의 수단으로 신중하게 적용해야 한다.

실무에서 사용하는 역정규화 기법

역정규화는 정해진 공식이 있는 것이 아니라, 현재 시스템이 겪고 있는 성능 문제의 종류에 따라 다양한 기법을 적용할 수 있다.

1. 중복 컬럼 추가

가장 흔하게 사용되는 기법이다. `JOIN`을 줄이기 위해 조회 시 자주 필요한 다른 테이블의 컬럼을 그대로 복사해서 가져오는 것이다.

- **문제 상황**: 주문 내역 페이지에서 각 주문 상품의 '상품명'을 보여주어야 한다. 이를 위해 매번 `order_item`과 `product` 테이블을 `JOIN` 해야 해서 성능이 저하된다.
- **해결책**: `order_item` 테이블에 `product_name` 컬럼을 추가한다.

```sql
-- 기존 order_item 테이블에 product_name 컬럼 추가
ALTER TABLE order_item ADD COLUMN product_name VARCHAR(100);

-- 주문이 들어올 때, product 테이블에서 상품명을 복사해서 저장
INSERT INTO order_item (order_id, product_id, order_price, order_quantity,
product_name)
VALUES (1004, 10, 1500000, 1, '노트북'); -- product_name을 직접 저장
```

- **결과**: 이제 상품명을 조회하기 위해 `product` 테이블을 `JOIN` 할 필요가 없어진다.

```sql
-- JOIN 없이 order_item 테이블만으로 상품명 조회 가능
SELECT order_id, product_name, order_quantity, order_price
FROM order_item
WHERE order_id = 1004;
```

- **주의할 점**: 이 방식의 치명적인 단점은 데이터 불일치 가능성이다. 만약 `product` 테이블에서 '노트북'의 이름이 '게이밍 노트북'으로 변경된다면, `order_item` 테이블에 이미 저장된 '노트북'이라는 이름은 자동으로 바뀌지 않는다.

- **실무 팁**: 주문 당시의 상품명, 가격 등은 '역사적 데이터'로 취급하여 변경하지 않는 것이 비즈니스 규칙상 타당할 수 있다. 이런 경우에는 중복 컬럼 추가가 매우 효과적인 해결책이 된다. 만약 데이터 변경을 전파해야 한다면, 애플리케이션 로직이나 데이터베이스 트리거(Trigger)를 통해 `product` 이름이 바뀔 때 관련된 모든 `order_item`을 업데이트해주는 추가적인 개발이 필요하다.

2. 파생 컬럼 추가 (계산된 값의 저장)

조회 시점에 복잡한 계산(SUM, COUNT 등)이 필요하여 부하가 발생하는 경우, 그 계산 결과를 미리 컬럼에 저장해 두는 방식이다.

- **문제 상황**: 마케팅 팀에서 VIP 고객을 선정하기 위해 회원별 '총 주문 금액'을 자주 조회한다. 이 값을 얻으려면 매번 특정 회원의 모든 `order_item`을 읽어 `order_price`와 `order_quantity`를 곱한 값을 모두 더해야 한다.
- **해결책**: `member` 테이블에 `total_purchase_amount` 라는 컬럼을 추가하고, 주문이 완료될 때마다 이 값을 업데이트한다.

```sql
-- member 테이블에 총 주문 금액 컬럼 추가
ALTER TABLE member ADD COLUMN total_purchase_amount INT NOT NULL DEFAULT 0;
```

- **결과**: 이제 회원별 총 주문 금액을 조회할 때 복잡한 집계 쿼리 없이 `member` 테이블에서 바로 값을 읽을 수 있다.

```sql
-- 복잡한 계산 없이 즉시 조회 가능
SELECT name, total_purchase_amount FROM member WHERE member_id = 1;
```

- **주의할 점**: 이 기법은 쓰기(Write) 작업의 부하를 증가시킨다. 새로운 주문이 들어올 때마다, 주문이 취소될 때마다, 반품이 발생할 때마다 `member` 테이블의 `total_purchase_amount` 값을 정확하게 갱신하는 로직을 애플리케이션에 반드시 구현해야 한다. 이 로직에 실수가 생기면 데이터는 쉽게 깨져버린다.

3. 테이블 통합 및 분할

테이블 간의 관계, 그리고 데이터의 사용 패턴을 분석하여 테이블 구조를 재조정하는 방법이다.

- **테이블 통합**: 1:1 또는 1:N 관계에서 항상 함께 조회되는 테이블들을 하나의 테이블로 합쳐 `JOIN`을 원천적으로 제거한다. (예: `member`와 `member_detail` 테이블을 하나로 합치는 경우)
- **테이블 분할**: 하나의 테이블에 컬럼이 너무 많고, 일부 컬럼만 자주 사용될 때 테이블을 수직으로 분할(`Vertical Partitioning`)하여 디스크 I/O 성능을 높일 수 있다. 자주 사용하는 컬럼들과 그렇지 않은 컬럼들을 별도의 테이블로 분리하는 것이다.

역정규화 시 데이터 일관성 유지 방안

역정규화의 가장 큰 대가는 **데이터 불일치(Inconsistency) 위험**이다. 원본 데이터가 변경되었을 때, 중복된 데이터를 빠짐없이 찾아서 업데이트해주어야 한다. 이 일관성을 유지하는 방법은 다음과 같다.

1. **애플리케이션 로직**: 가장 일반적인 방법이다. 개발자가 코드 레벨에서 데이터 일관성을 책임지는 것이다. 상품명을 수정하는 로직을 구현할 때, `product` 테이블을 `UPDATE` 한 후, 관련된 `order_item` 테이블의 `product_name`도 `UPDATE` 하는 코드를 반드시 함께 작성해야 한다.
2. **데이터베이스 트리거(Trigger)**: 특정 테이블에 `INSERT`, `UPDATE`, `DELETE` 같은 이벤트가 발생했을 때, 미리 정의해둔 다른 SQL을 자동으로 실행시키는 데이터베이스 기능이다. 예를 들어, `product` 테이블에 `UPDATE` 트리거를 설정하여, `product_name` 컬럼이 변경되면 `order_item` 테이블의 `product_name`을 자동으로 업데이트하게 만들 수 있다.

3. **배치(Batch) 작업**: 주기적으로(예: 매일 새벽) 스케줄러를 통해 배치 프로그램을 실행시켜, 원본 데이터와 중복된 데이터 간의 불일치를 찾아내고 동기화시켜주는 방법이다. 실시간 일관성이 중요하지 않은 데이터에 사용할 수 있다.

역정규화는 성능 문제를 해결하는 강력한 '양날의 검'이다. 반드시 필요한 곳에, 데이터 일관성을 유지할 수 있는 명확한 계획을 가지고 적용해야 한다.

역정규화, 언제 해야 할까? (실무 가이드)

역정규화는 양날의 검과 같다. 잘못 사용하면 데이터 무결성을 해치고 시스템을 더 복잡하게 만드는 재앙이 될 수 있다. 실무에서는 다음 원칙을 반드시 지켜야 한다.

1. **섣불리 적용하지 마라**: 설계 단계에서부터 역정규화를 고려하는 것은 매우 위험하다. **먼저 정규화 원칙에 따라 데이터 모델을 설계하는 것이 기본이다.**
2. **데이터로 증명하라**: '느릴 것 같다'는 추측만으로 역정규화를 진행해서는 안 된다. 반드시 서비스 오픈 전 성능 테스트 또는 애플리케이션을 운영하면서 **실제 성능 측정을 통해 병목이 되는 쿼리를 명확히 식별**해야 한다.
3. **읽기/쓰기 비율을 고려하라**: 역정규화는 주로 조회(읽기) 성능을 위해 쓰기 성능과 데이터 정합성을 희생하는 구조다. 따라서 **쓰기 작업보다 읽기 작업이 압도적으로 많은 경우**에 적용하는 것이 효과적이다.
4. **비용을 계산하라**: 역정규화를 통해 얻는 성능적 이점과, 그로 인해 발생하는 데이터 불일치 문제 해결 및 유지보수를 위한 개발 비용을 철저히 비교 분석해야 한다.

결론적으로, **역정규화는 정규화된 모델에서 시작하여, 실제 운영 환경에서 발생한 성능 문제를 해결하기 위한 최후의 수단**으로 사용되어야 한다. 정규화의 원칙을 깊이 이해하고, 그 위에 시스템의 특성을 고려하여 역정규화를 전략적으로 적용하는 것이 바로 뛰어난 데이터베이스 설계자의 역량이다.

역정규화는 언제 적용하는가?

데이터베이스 설계는 크게 개념적 모델링, 논리적 모델링, 물리적 모델링의 3단계로 진행된다. 우리가 지금까지 배운 정규화는 데이터의 논리적 구조를 정의하는 **논리적 모델링 단계**의 핵심 과정이다.

반면, 역정규화는 논리적 모델이 완성된 후, 실제 데이터베이스의 성능, 저장 공간 등을 고려하여 물리적인 저장 구조를 설계하는 **물리적 모델링 단계**에서 적용하는 기법이다.

이 순서가 매우 중요하다. 반드시 정규화를 통해 데이터의 정합성과 일관성을 확보한 설계를 먼저 완성해야 한다. 그 이후에, 성능 저하가 예상되는 특정 부분에 한해 역정규화를 신중하게 검토하고 적용해야 한다. 처음부터 역정규화를 염두에 두고 설계하면 데이터 구조의 무결성이 깨지고, 결국 유지보수하기 어려운 복잡한 시스템이 될 위험이 크다. **'선 정규**

화, 후 역정규화' 라는 원칙을 반드시 기억해야 한다.

> **용어 정리: 역정규화, 반정규화, 비정규화**
> 실무에서는 Denormalization이라는 용어를 역정규화, 반정규화, 비정규화 등으로 번역하여 사용하며, 엄격한 구분 없이 혼용하는 경우가 많다. 그중에서도 주로 '반정규화'와 '역정규화'라는 단어가 많이 사용된다. 각각의 미묘한 차이를 알아보자.
>
> - **반정규화**: '정규화에 반대한다'는 직관적인 의미를 가진다. 가장 포괄적인 용어이지만, 기술적인 정밀성은 다소 떨어진다.
> - **역정규화**: '역'이라는 단어가 의미하듯, **이미 정규화된 모델을 성능 등의 이유로 의도적으로 되돌리는 '과정(Process)'**을 표현하는 용어다. 우리는 정규화된 모델에서 출발하여 성능 최적화를 위해 의도적으로 중복을 추가하는 작업을 할 것이므로, 이 강의에서는 '역정규화'라는 용어를 사용한다.
> - **비정규화**: 정규화가 전혀 적용되지 않은 **상태**나 구조를 의미한다. 설계 초기 단계에서 정규화를 거치지 않았거나, NoSQL처럼 처음부터 정규화 원칙을 따르지 않는 데이터 모델을 설명할 때 적합한 용어다.

테이블 정의서

물리적 모델링의 최종 산출물 중 하나는 바로 **테이블 정의서(Table Definition Document)**다. 테이블 정의서는 데이터베이스의 각 테이블과 컬럼이 무엇을 의미하고, 어떤 규칙을 가지고 있는지 상세하게 설명하는 '설계 명세서'와 같다.

테이블 정의서, 왜 필요한가?

문제 상황: 당신은 새로운 프로젝트에 투입된 개발자다. 데이터베이스 스키마를 파악하기 위해 DDL(`CREATE TABLE ...`) 코드를 열어보았다.

```
CREATE TABLE order_item (
    oi_id   BIGINT NOT NULL AUTO_INCREMENT,
    o_id    BIGINT NOT NULL,
```

```
    p_id    BIGINT NOT NULL,
    price   INT    NOT NULL,
    qty     INT    NOT NULL,
    PRIMARY KEY (oi_id)
);
```

이 DDL만 보고 다음 질문에 바로 답할 수 있는가?

- `order_item` 테이블은 정확히 어떤 데이터를 저장하는 테이블인가?
- `oi_id`, `o_id`, `p_id`는 각각 무엇의 ID를 의미하는가?
- `price`는 상품의 현재 가격인가, 아니면 주문 당시의 가격인가?
- `qty`는 무슨 수량을 의미하는가?

아마 대부분의 사람은 정확히 답하기 어려울 것이다. 이처럼 DDL은 데이터베이스의 물리적인 구조는 알려주지만, 각 요소가 담고 있는 **비즈니스적 의미**나 **숨겨진 규칙**까지 설명해주지는 않는다.

이러한 정보의 공백을 메워주는 것이 바로 테이블 정의서다. 잘 작성된 테이블 정의서는 DDL만으로는 알 수 없는 풍부한 맥락을 제공하여, 개발자나 데이터 분석가가 데이터의 구조와 의미를 명확하게 이해할 수 있도록 돕는다.

테이블 정의서의 역할

테이블 정의서는 단순한 문서가 아니라, 프로젝트의 성공에 기여하는 중요한 자산이다.

1. **의사소통의 중심**: 개발자, DBA, 기획자, 데이터 분석가 등 프로젝트에 참여하는 모든 사람이 데이터에 대해 동일한 이해를 갖도록 돕는 **'공통 언어'** 역할을 한다. "이 컬럼은 어떤 의미인가요?" 와 같은 불필요한 질문과 오해를 줄여준다.
2. **유지보수의 핵심**: 시간이 흘러 시스템이 복잡해지고 담당자가 바뀌더라도, 테이블 정의서만 보면 누구나 데이터 구조를 빠르게 파악하고 유지보수 작업을 수행할 수 있다.
3. **데이터 품질 보증**: 각 컬럼의 제약 조건, 기본값, 허용되는 값의 범위 등을 명시함으로써 데이터의 일관성과 품질을 유지하는 데 기여한다.
4. **데이터베이스의 청사진**: 이 문서만 보면 누구나 데이터베이스의 전체 구조를 이해하고, 테이블을 생성(CREATE)할 수 있어야 한다.
5. **개발의 가이드**: 개발자는 이 문서를 보고 테이블과 컬럼의 이름을 정확히 사용하여 SQL을 작성하고, 애플리케이션 코드를 구현한다.

테이블 정의서의 구성 요소

테이블 정의서의 형식은 회사나 팀마다 다를 수 있지만, 일반적으로 다음과 같은 필수 정보들을 포함한다.

1. 테이블 정보

- **테이블 한글명 (논리명)**: 테이블이 무엇을 나타내는지 비즈니스 용어로 설명한다. (예: 회원)
- **테이블 영문명 (물리명)**: 실제 데이터베이스에 생성되는 테이블의 이름이다. (예: `member`)
- **테이블 설명**: 테이블의 역할과 목적을 상세하게 기술한다.

2. 컬럼 정보

- **No.**: 컬럼의 순번
- **컬럼 한글명 (논리명)**: 컬럼의 비즈니스적 의미를 설명한다. (예: 회원ID, 로그인ID)
- **컬럼 영문명 (물리명)**: 실제 테이블에 생성되는 컬럼의 이름이다. (예: `member_id`, `login_id`)
- **데이터 타입**: `VARCHAR(100)`, `BIGINT`, `DATETIME` 등 컬럼의 데이터 타입을 명시한다.
- **제약 조건 (Key)**: 기본 키(PK), 외래 키(FK), 고유 키(Unique) 여부를 표시한다.
- **NULL 허용 여부**: `NOT NULL`(필수값) 또는 `NULL`(선택값) 여부를 표시한다.
- **기본값 (Default)**: 컬럼에 기본으로 설정된 값이 있다면 명시한다.
- **비고 (설명)**: 컬럼에 대한 추가적인 설명이나 규칙을 자유롭게 기술한다. (예: '비밀번호는 암호화해서 저장', '주문 당시의 가격을 기록')

쇼핑몰 테이블 정의서 예시

우리가 설계한 `member` 테이블을 가지고 실제 테이블 정의서를 작성해 보자.

테이블 정의: member

테이블 한글명	회원
테이블 영문명	`member`
테이블 설명	쇼핑몰에 가입한 회원의 기본 정보를 저장하는 테이블

컬럼 정의

No.	컬럼 한글명	컬럼 영문명	데이터 타입	제약 조건	기본값	비고
1	회원 ID	`member_id`	`BIGINT`	PK	`AUTO_INCREMENT`	회원의 고유 식별자 (대리 키)

2	로그인 ID	`login_id`	`VARCHAR(50)`	UQ		회원이 로그인 시 사용하는 ID
3	비밀번호	`password`	`VARCHAR(255)`			비밀번호 (반드시 암호화해서 저장)
4	회원명	`member_name`	`VARCHAR(50)`			회원의 실명
5	이메일	`email`	`VARCHAR(100)`	UQ		회원 인증 및 소통을 위한 이메일
6	주소	`addr`	`VARCHAR(255)`	NULL		상품 배송을 위한 주소
7	가입일	`created_at`	`DATETIME`		`CURRENT_TIMESTAMP`	회원 가입 시점
8	수정일	`updated_at`	`DATETIME`		`CURRENT_TIMESTAMP`	데이터 수정 시 자동 갱신

- 표를 간결하게 표현하기 위해 NULL 허용을 제약조건에 함께 포함했다. NULL 허용은 별도의 컬럼으로 분리하는 것이 좋다.
- `addr` 축약어 합의

이처럼 테이블 정의서를 작성하면, DDL만 봤을 때는 알기 어려웠던 각 컬럼의 정확한 의미와 규칙(예: 비밀번호는 암호화해서 저장해야 함)을 명확하게 알 수 있다.

> **실무 팁**
> 테이블 정의서는 한 번 만들고 끝나는 문서가 아니다. 데이터베이스의 스키마가 변경될 때마다 반드시 함께 업데이트되어야 하는 **'살아있는 문서(Living Document)'**다.
> 실무에서는 Excel, Google Sheets, 또는 Confluence나 Notion과 같은 위키 시스템을 사용하여 테이블 정의서를 작성하고 팀원들과 공유하며 관리한다.

이제 우리는 논리적 모델을 물리적 테이블로 변환하는 규칙을 배우고, 각 컬럼의 데이터 타입을 정했으며, 그 결과를 테이블 정의서라는 최종 설계도로 문서화하는 방법까지 알게 되었다. 이로써 데이터베이스 설계의 이론적인 부분은 모두 마무리되었다. 다음 장에서는 이 모든 이론을 종합하여 우리 손으로 직접 쇼핑몰 데이터베이스를 구축하는 실습을 진행해 보자.

정리

물리적 모델링 개요

- 물리적 모델링은 논리적 모델을 특정 DBMS(예: MySQL)에 맞게 변환하여 성능, 저장 공간, 유지보수 등을 최적화하는 과정이다.
- 논리적 모델링이 데이터의 정합성에 초점을 맞춘다면, 물리적 모델링은 실제 시스템의 **성능**에 초점을 맞춘다.
- 주요 작업에는 테이블/컬럼 영문명 변환, 데이터 타입 정의, 인덱스 설계, 역정규화 수행 등이 포함된다.

테이블과 컬럼 변환 규칙1 - 기본 규칙

- 프로젝트의 일관성과 가독성을 위해 명명 규칙(Naming Convention)을 정하고 따라야 한다.
- **일반 규칙**: 영어 사용, 소문자 스네이크 케이스(`snake_case`), 명확한 이름 사용, 예약어 사용 금지를 권장한다.
- **컬럼 규칙**:
  - 기본 키(PK)는 `테이블명_id` 형식으로 명명한다. (예: `member_id`)
  - 외래 키(FK)는 참조하는 테이블의 PK명을 그대로 사용한다. (예: `orders` 테이블의 `member_id`)
  - 날짜/시간 컬럼은 용도에 따라 접미사(`_at`, `_dt`, `_date`)를 붙여 의미를 명확히 한다.
  - 불리언(Boolean) 타입 컬럼은 `is_` 또는 `has_` 접두사를 사용한다.

테이블과 컬럼 변환 규칙2 - 축약어와 단수 복수

- 의미가 불분명한 축약어는 유지보수 비용을 증가시키므로, 명확성을 위해 전체 단어를 사용하는 것을 원칙으로 한다.
- `id`(identifier)처럼 보편적이고 모호하지 않은 축약어는 사용을 권장한다.
- 테이블 이름은 엔티티의 집합을 의미하는 복수형(`members`)과 엔티티의 틀을 의미하는 단수형(`member`) 중 하나를 선택하되, **프로젝트 내에서 일관성을 유지하는 것이 가장 중요하다.**
- 한글 논리명을 영문 물리명으로 변환할 때의 혼란을 막기 위해, 설계 초기에 **용어 사전**을 만들어 표준을 정하는 것이 매우 효과적이다.

데이터 타입1 - 문자, 숫자, PK 타입

- 데이터 타입을 잘못 선택하면 저장 공간 낭비, 성능 저하, 데이터 무결성 훼손 등의 문제가 발생할 수 있다.
- **문자열**: 대부분의 경우 가변 길이인 `VARCHAR`를 사용한다. 길이는 저장될 데이터를 고려하여 합리적으로 설정해야 한다.
- **숫자**: 금융 정보처럼 정확한 계산이 필요하면 `DECIMAL`을, 일반적인 ID나 개수는 `INT` 또는 `BIGINT`를 사용한다.
- **PK 타입**: 데이터가 폭발적으로 증가할 가능성이 있는 핵심 테이블(회원, 주문 등)의 PK는 미래 확장성과 관리의

편의성을 위해 `BIGINT` 사용을 권장한다. `INT`와 `BIGINT`의 저장 공간 차이는 현대 스토리지 환경에서 미미하다.

데이터 타입2 - 날짜와 시간 타입

- 날짜와 시간을 저장할 때는 `DATE`, `DATETIME`, `TIME` 같은 전용 타입을 사용해야 날짜 계산 및 비교가 용이하다.
- 실무에서는 대부분의 테이블에 데이터의 생성 및 수정 이력을 추적하기 위해 `created_at`(생성일시)과 `updated_at`(수정일시) 컬럼을 추가한다.
- `DEFAULT CURRENT_TIMESTAMP`와 `ON UPDATE CURRENT_TIMESTAMP` 옵션을 사용하면, 개발자가 신경 쓰지 않아도 데이터베이스가 생성 및 수정 시간을 자동으로 관리해주어 매우 편리하고 실수를 방지한다.

역정규화

- 조회 성능 향상을 위해 **의도적으로 정규화 원칙을 위배**하여 데이터의 중복을 허용하는 과정이다.
- 잦은 `JOIN`으로 인해 성능 저하가 발생할 때 사용하며, 읽기 성능을 높이는 대신 데이터 불일치 위험과 쓰기 성능 저하를 감수해야 하는 트레이드오프 관계다.
- 주요 기법으로는 `JOIN`을 줄이기 위한 **중복 컬럼 추가**, 복잡한 계산을 미리 해두는 **파생 컬럼 추가**, **테이블 통합** 등이 있다.
- **'선 정규화, 후 역정규화'** 원칙을 따라야 하며, 추측이 아닌 실제 성능 테스트를 통해 병목 지점이 확인되었을 때 최후의 수단으로 신중하게 적용해야 한다.

테이블 정의서

- 데이터베이스의 각 테이블과 컬럼의 **비즈니스적 의미, 규칙, 제약조건** 등을 상세히 기술한 설계 명세서다.
- 개발자, 기획자 등 모든 프로젝트 참여자가 데이터에 대해 동일한 이해를 갖도록 돕는 **의사소통의 중심** 역할을 한다.
- 테이블의 논리명/물리명, 컬럼의 논리명/물리명, 데이터 타입, 제약 조건, NULL 허용 여부, 비고 등의 정보를 포함한다.
- 스키마가 변경될 때마다 함께 업데이트되어야 하는 **'살아있는 문서(Living Document)'**로 관리해야 한다.