12. 물리적 모델링 - 실습

#0.강의/2.데이터베이스로드맵/3.설계1

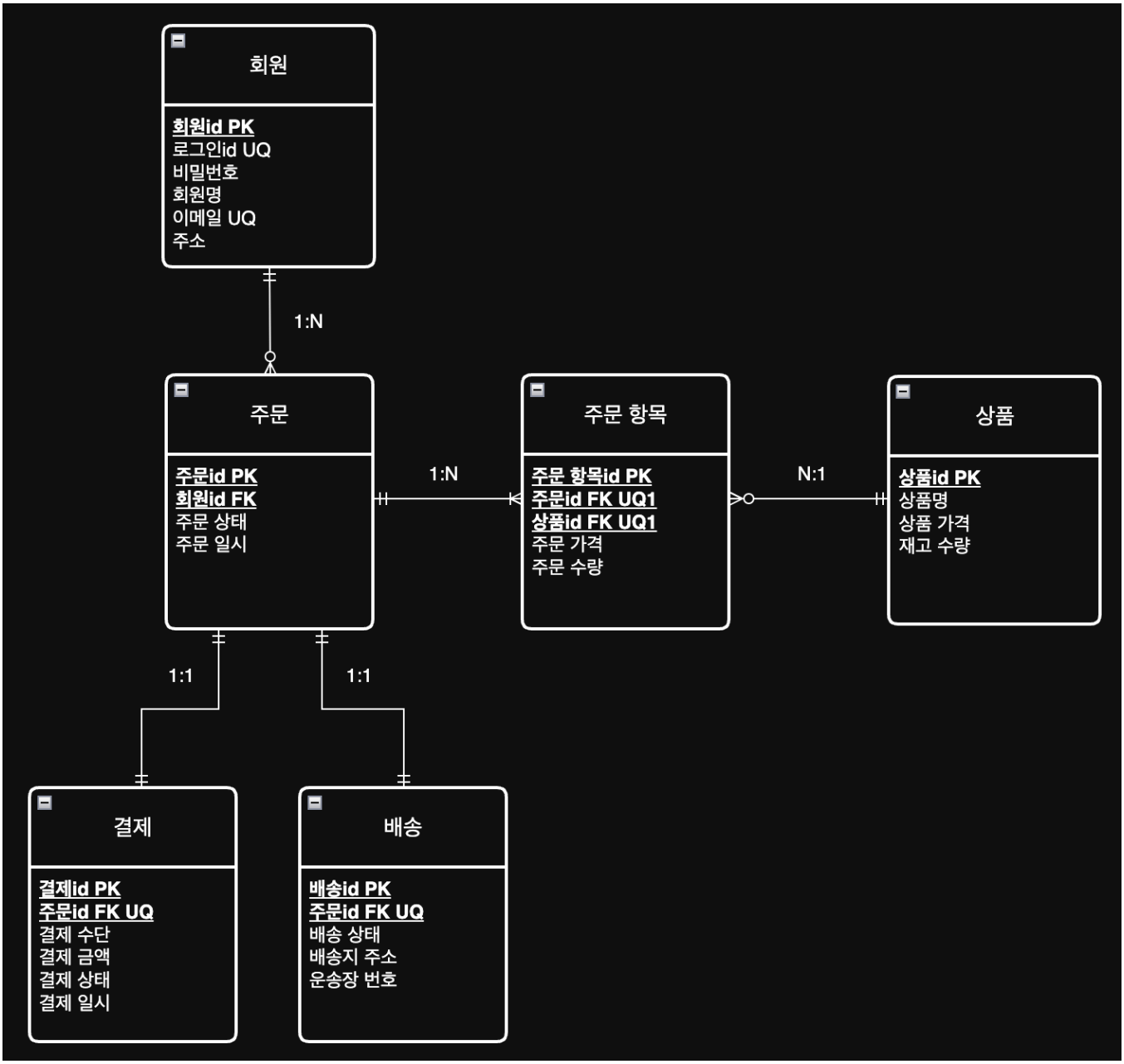
물리적 모델링 - 실습 시작

드디어 데이터베이스 설계의 마지막 여정이다. 우리는 요구사항 분석에서 시작해 개념적 모델링(ERD), 논리적 모델링(정규화)을 거쳐, 물리적 모델링의 구체적인 기법들까지 모두 학습했다.

지금까지 배운 모든 지식을 총동원하여, 우리의 쇼핑몰 논리적 모델을 실제 MySQL 데이터베이스에서 동작하는 물리적 스키마로 변환하는 최종 실습을 진행해보자. 우리가 직접 DDL(`CREATE TABLE`)을 작성하여 살아 숨 쉬는 데이터베이스를 만드는 과정이다.

논리적 모델링 확인

물리적 모델링을 시작하기 전에 앞서 진행한 논리적 모델링을 다시 확인해보자.



논리적 모델링

앞서 개념적 모델을 바탕으로 데이터의 논리적 구조와 관계를 정의하는 논리적 모델링을 완성했다. 우리는 대리 키와 비식별 관계를 사용하는 현대적인 설계 원칙에 따라, 정규화된 아주 깔끔한 모델을 만들었다.

이제 이 논리적 모델을 우리가 사용하기로 한 MySQL이라는 특정 데이터베이스에 맞게 변환하고, 성능과 효율성, 그리고 현실적인 제약 조건들을 고려하여 최적화하는 **물리적 모델링**을 진행할 차례다.

물리적 모델링

물리적 모델링은 각 테이블별로 어떤 데이터 타입을 선택하고, 어떤 제약조건과 인덱스를 추가해야 할지 신중하게 결정하며 `CREATE TABLE` 구문을 완성해 나간다.

물리적 모델링의 결과는 크게 **테이블 정의서**와 **실제 테이블을 만들 수 있는 스크립트인 DDL**이 나와야 한다.

물리적 모델링의 핵심 목표 중 하나는 바로 **'성능'**이다. 논리적으로 아무리 완벽한 모델이라도, 실제 서비스에서 쿼리가 느리다면 아무 소용이 없다. 이번 시간에는 조회 성능을 극대화하기 위해 의도적으로 정규화 원칙을 위배하는 **역정규화(Denormalization)** 기법과 데이터 검색 속도를 획기적으로 높이는 **인덱스(Index)** 설계를 우리의 쇼핑몰 모델에 직접 적용해 볼 것이다.

물리적 모델링 순서

앞서 설명했듯이 물리적 모델링은 일반적으로 다음과 같은 순서를 따른다.

1. **테이블과 컬럼 변환**: 논리적 모델의 엔티티는 테이블로, 속성은 컬럼으로 변환한다. 이때 테이블명, 컬럼명을 정하는 규칙(Naming Convention)을 정하고 적용한다.
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.

여기서 1,2,3 은 앞서 물리적 모델링에서 설명한 내용대로 진행하면 된다. 여기서는 4. 인덱스 설계와 5. 역정규화 및 성능 튜닝에 집중하겠다.

> ✨ **용어 사전 활용**
> 테이블과 컬럼을 변환할 때는 용어 사전을 적극 활용하자.

> **뷰, 저장프로시저, 함수, 트리거**
> 뷰, 저장프로시저, 함수, 트리거는 여기서는 사용하지 않겠다. 자세한 내용은 실전 데이터베이스 기본편을 참고하자.

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

인덱스 설계 - 실습

인덱스는 데이터베이스 테이블의 특정 컬럼(들)의 데이터를 빠르게 찾을 수 있도록 도와주는 일종의 '찾아보기'나 '목차'와 같다. 인덱스가 없으면, 데이터베이스는 원하는 데이터를 찾기 위해 테이블의 모든 행을 처음부터 끝까지 스캔해야 한다. 이를 **풀 테이블 스캔(Full Table Scan)**이라고 한다. 데이터가 수백만, 수천만 건이 되면 이는 재앙과도 같은 성능 저하를 일으킨다.

"우리 쇼핑몰의 어떤 기능이 인덱스를 필요로 할까?"

이 질문에 답하기 위해, 우리 쇼핑몰에서 매우 자주 발생할 것으로 예상되는 핵심적인 조회 시나리오들을 분석하고, 그에 맞는 인덱스를 전략적으로 추가해 보자.

> **MySQL 인덱스 자동 생성**
> MySQL에서는 **Primary Key(PK)**와 **Unique Key(UK)**, **Foreign Key(FK)**를 설정하면 자동으로 해당 컬럼에 인덱스가 생성된다.

시나리오 1: 로그인 및 중복 ID, 이메일 체크

사용자가 로그인을 시도하거나, 회원가입 시 ID나 이메일이 중복되는지 확인해야 한다. 이 작업은 매우 빈번하게 일어난다.

```
-- 로그인 시
SELECT member_id, password FROM member WHERE login_id = 'user123';

-- 이메일로 회원 정보 찾기
SELECT member_id FROM member WHERE email = 'user123@example.com';
```

문제점: `member` 테이블에 회원이 100만 명 있다면, 이 쿼리는 100만 개의 데이터를 모두 스캔(Full Table Scan)해야 한다.

해결책: `login_id`와 `email` 컬럼은 UNIQUE 제약조건을 걸어두었는데, MySQL에서 UNIQUE 제약조건은 자동으로 해당 컬럼에 고유 인덱스(Unique Index)를 생성한다. 따라서 이 경우는 이미 최적화가 되어 있지만, 만약 UNIQUE가 아닌 일반 컬럼이었다면 반드시 인덱스를 생성해야 한다.

시나리오 2: 회원의 주문 목록 조회

사용자가 '마이페이지'에서 자신의 과거 주문 내역을 조회한다. 이는 쇼핑몰에서 가장 빈번하게 일어나는 조회 중 하나다.

```
-- member_id가 10번인 회원의 모든 주문을 최신순으로 조회한다.
SELECT *
FROM orders
WHERE member_id = 10
ORDER BY ordered_at DESC;
```

이 기능은 `orders` 테이블에서 특정 `member_id`를 가진 주문들을 찾는 쿼리를 실행하게 된다.

인덱스 필요성: `member_id` 컬럼에 인덱스가 없다면, MySQL은 이 쿼리를 처리하기 위해 `orders` 테이블의 모든 데이터를 다 뒤져서 `member_id`가 10인 것을 찾아내야 한다. `orders` 테이블에 데이터가 쌓일수록 이 작업은 점점 더 느려질 것이다. 따라서 `WHERE` 절에서 자주 사용되는 `member_id` 컬럼에는 반드시 인덱스가 필요하다.

인덱스 추가

```
-- orders 테이블의 member_id 컬럼에 인덱스를 추가한다.
CREATE INDEX idx_member_id ON orders (member_id);
```

> **실무 팁**
> MySQL에서는 외래 키(Foreign Key)를 생성하면 해당 컬럼에 **자동으로 인덱스가 생성**된다. 따라서 `orders.member_id`는 사실 FK 제약조건을 거는 순간 인덱스가 이미 만들어진다.
> 하지만 FK 제약조건을 사용하지 않는 경우 반드시 인덱스를 직접 추가해야 한다.

시나리오 3: 상품명으로 상품 검색

사용자가 쇼핑몰 상단의 검색창에 '노트북'이라고 입력하여 상품을 검색한다.

```
-- 이름에 '노트북'이 포함된 상품을 찾는다.
SELECT *
FROM product
WHERE product_name LIKE '노트북%';
```

이 기능은 `product` 테이블의 `product_name` 컬럼에서 특정 키워드를 포함하는 상품을 찾는 `LIKE` 쿼리를 사용하게 된다.

인덱스 필요성: 상품 검색 역시 매우 중요한 기능이다. `name` 컬럼에 인덱스가 없다면, 검색할 때마다 전체 상품을 스캔해야 하므로 매우 느린 검색 경험을 제공하게 된다.

인덱스 추가

```
-- product 테이블의 name 컬럼에 인덱스를 추가한다.
CREATE INDEX idx_product_name ON product (product_name);
```

> ⚠ 주의
>
> `LIKE '%검색어%'` 처럼 검색어 앞에 `%`가 붙는 경우에는 일반적인 B-Tree 인덱스가 제대로 동작하지 않아 성능 향상에 한계가 있다. 이런 전문적인 텍스트 검색 기능을 위해서는 MySQL의 `Full-Text Index`나 `Elasticsearch` 같은 별도의 검색 엔진을 사용하는 것이 실무적인 해결책이다. 하지만 `LIKE '검색어%'`와 같이 검색어 앞에 `%`가 없는 경우에는 인덱스가 효과적으로 사용된다.

시나리오 4: 관리자의 주문 상태 및 기간별 조회 (복합 인덱스)

쇼핑몰 관리자가 어드민 페이지에서 '지난 한 달간 취소된 주문(`CANCELED`)' 목록을 조회하여 CS 처리를 하려고 한다.

```
-- 2025년 7월 한 달간 취소된 주문을 조회한다.
SELECT *
FROM orders
WHERE order_status = 'CANCELED'
```

```
    AND ordered_at BETWEEN '2025-07-01 00:00:00' AND '2025-07-31 23:59:59';
```

이 기능은 `orders` 테이블에서 `order_status`와 `ordered_at` 두 가지 조건을 동시에 만족하는 데이터를 찾는다.

> **BETWEEN 쿼리와 밀리초 문제**
> 여기서는 예제를 단순하고 쉽게 설명하기 위해 일시에 BETWEEN 쿼리를 사용했지만, 실무에서는 이렇게 사용할 때 주의할 점이 있다.
> 만약 `ordered_at` 컬럼에 '2025-07-31 23:59:59.500' 과 같이 초 단위보다 더 상세한 시간이 저장되어 있다면, 이 데이터는 '2025-07-31 23:59:59' 보다 크다고 판단되어 쿼리 결과에서 누락된다.
>
> 밀리초가 사용되는 경우 가장 좋은 해결책은 다음과 같이 **시작일은 포함(>=)하고, 종료일의 다음 날은 미포함(<)**하는 방식을 사용하면 된다.
>
> ```
> SELECT *
> FROM orders
> WHERE order_status = 'CANCELED'
>  AND ordered_at >= '2025-07-01'
>  AND ordered_at < '2025-08-01'; -- 8월 1일 0시 0분 0초보다 작은 데이터 조회
> ```

인덱스 필요성: `order_status`와 `ordered_at` 각각에 단일 인덱스가 있다면, 데이터베이스 옵티마이저는 둘 중 더 효율적이라고 판단되는 인덱스 하나만 사용하게 된다. 이는 여전히 비효율적일 수 있다. 두 개 이상의 컬럼이 `WHERE` 절 조건으로 함께 자주 사용될 때는, 이 컬럼들을 묶어서 **복합 인덱스(Composite Index)**를 만들어주는 것이 훨씬 더 효율적이다.

복합 인덱스 추가

```
-- orders 테이블에 order_status와 ordered_at를 묶는 복합 인덱스를 추가한다.
CREATE INDEX idx_status_ordered_at ON orders (order_status, ordered_at);
```

중요: 복합 인덱스의 컬럼 순서
복합 인덱스는 컬럼의 순서가 매우 중요하다. 인덱스는 첫 번째 컬럼 기준으로 정렬되고, 그 안에서 두 번째 컬럼 기준으로 정렬되는 방식으로 동작한다. 따라서 조회 조건에서 **선택도(Selectivity)가 높고, `=`(등호) 조건으로 사용되는 컬럼을 앞에 두는 것**이 일반적인 원칙이다.

- `order_status`: 'ORDERED', 'CANCELED', 'SHIPPING' 등 몇 가지 값으로 정해져 있어 선택도가 낮다. 하지만 `=`(등호) 조건으로 사용된다.
- `ordered_at`: 값의 종류가 매우 다양하여 선택도가 높다. 하지만 `BETWEEN` 같은 범위 조건으로 사용된다.
- 이 경우, `=`(등호) 조건으로 범위를 좁힐 수 있는 `order_status`를 앞에 두고, 그 안에서 `ordered_at`로 원하는 범위를 빠르게 찾도록 `(order_status, ordered_at)` 순서로 인덱스를 구성하는 것이 효과적이다.

> **복합 인덱스 대원칙**
>
> 복합 인덱스를 설계하고 사용할 때는 다음 세 가지 대원칙을 반드시 기억하자!
>
> 1. **인덱스는 순서대로 사용하라!** (왼쪽 접두어 규칙)
> 2. **등호(=) 조건은 앞으로, 범위 조건(<, >)은 뒤로!**
> 3. **정렬(ORDER BY)도 인덱스 순서를 따르라!**
>
> 복합 인덱스에서 등호 조건은 앞으로, 범위 조건은 뒤로 설정해야 한다. 자세한 내용은 실전 데이터베이스 기본편 - 복합 인덱스를 참고하자.

시나리오 5: 사용자의 주문 상태 및 기간별 조회

일반 사용자가 마이페이지에서 자신의 주문 중 '배송 완료' 상태인 것을 최근 3개월치만 모아보고 싶어한다.

```sql
-- member_id가 10번인 회원의 최근 3개월간 '배송 완료(COMPLETED)'된 주문을 조회한다.
SELECT *
FROM orders
WHERE member_id = 10
  AND order_status = 'COMPLETED'
  AND ordered_at >= DATE_SUB(NOW(), INTERVAL 3 MONTH);
```

이 기능은 `orders` 테이블에서 특정 `member_id`를 가지면서, `order_status`와 `ordered_at` 조건까지 만족하는 데이터를 찾는다.
따라서 `(member_id, order_status, ordered_at)` 순서로 복합 인덱스를 고려할 수 있다.

그런데 이 경우에는 인덱스를 만들지 말지 고민이 필요하다.
앞선 관리자 시나리오와 매우 유사해 보이지만, 결정적으로 다른 점이 하나 있다. 바로 `WHERE` 절에 `member_id = 10` 조건이 포함된다는 것이다. 이 작은 차이가 인덱스 설계 전략을 완전히 바꾼다.

인덱스를 추가하지 않는 이유: 데이터의 선택도(Selectivity)와 비용 고려

데이터베이스는 쿼리를 실행할 때, **가장 효율적으로 데이터를 걸러낼 수 있는 인덱스를 먼저 사용**하려는 경향이 있다. 이것을 **'선택도가 높다'**라고 표현한다. 선택도는 특정 값으로 얼마나 많은 데이터를 걸러낼 수 있는지를 나타내는 지표다.

1. `member_id`의 압도적으로 높은 선택도

- `order_status = 'COMPLETED'` 조건: 전체 주문 데이터가 1억 건이라면, '배송 완료' 상태인 주문은 수천만 건에 달할 수 있다. 데이터를 크게 줄이지 못한다.
- `member_id = 10` 조건: 전체 1억 건의 주문 중에서 특정 회원 ID가 10번인 주문은 많아야 수십, 수백 건에 불과할 것이다. 이 조건 하나만으로 조회 대상이 1억 건에서 수백 건으로 극적으로 줄어든다.

따라서 MySQL 옵티마이저는 `member_id` 컬럼에 이미 생성되어 있는 인덱스(`idx_member_id` 또는 FK로 인해 자동 생성된 인덱스)를 사용하는 것이 압도적으로 효율적이라고 판단한다.

2. 인덱스 사용 후의 동작 과정

MySQL은 다음과 같은 순서로 이 쿼리를 처리한다.

1. `idx_member_id` 인덱스를 사용하여 `member_id`가 10인 주문 데이터가 디스크의 어디에 저장되어 있는지 빠르게 찾아낸다.
2. 찾아낸 수백 건의 데이터만 메모리로 가져온다.
3. 메모리에 올라온 이 소량의 데이터를 대상으로 `order_status = 'COMPLETED'`인지, `ordered_at`가 최근 3개월 이내인지 조건을 확인(필터링)한다.

수억 개의 전체 데이터를 스캔하는 것과 비교할 때, 이미 수백 개로 줄어든 데이터를 메모리에서 필터링하는 비용은 거의 무시할 수 있을 정도로 저렴하고 빠르다.

인덱스의 비용: 모든 쿼리에 인덱스를 만들지 않는 이유

만약 `(member_id, order_status, ordered_at)` 순서로 복합 인덱스를 만든다면 이 쿼리가 조금 더 빨라질 수는 있다. 하지만 약간의 성능 향상을 위해 우리가 감수해야 할 비용도 고려해야 한다.

- **저장 공간 비용**: 인덱스는 데이터가 아니다. 데이터를 빠르게 찾기 위한 별도의 자료구조이며, 디스크 공간을 차지한다. 인덱스를 많이 만들수록 데이터베이스의 전체 크기는 커진다.
- **쓰기(Write) 성능 저하**: 인덱스는 테이블의 데이터가 변경될 때마다 함께 업데이트되어야 한다.
    - `INSERT`: 새로운 주문이 들어오면 `orders` 테이블뿐만 아니라, 이 테이블에 속한 모든 인덱스에도 새로

운 정보를 추가해야 한다.

- `UPDATE`: 주문 상태가 'ORDERED'에서 'SHIPPING'으로 바뀌면, `(member_id, order_status, ordered_at)` 인덱스의 `order_status` 부분을 포함하여 정렬 순서를 다시 맞춰야 하는 복잡한 작업이 발생한다.
- `DELETE`: 주문이 삭제되면 모든 인덱스에서도 해당 데이터를 삭제해야 한다.

이처럼 인덱스는 조회(`SELECT`) 속도를 향상시키는 강력한 도구이지만, 데이터 변경(`INSERT`, `UPDATE`, `DELETE`) 작업에는 부하를 주는 양날의 검이다.

결론적으로, `member_id`라는 강력한 필터링 조건 덕분에 이미 충분히 빠른 쿼리에 추가 인덱스를 생성하는 것은, 얻는 이점(미미한 조회 성능 향상)보다 잃는 것(저장 공간 낭비와 심각한 쓰기 성능 저하)이 큰, 전형적인 '과잉 최적화(Over-optimization)'가 될 수 있다.

실무에서는 모든 조회 케이스에 인덱스를 거는 것이 아니라, **전체 시스템에 병목 현상을 일으키는 느린 쿼리(Slow Query)를 찾아내고, 그 쿼리들을 중심으로 가장 효율적인 인덱스를 전략적으로 설계**하는 것이 핵심이다.

인덱스를 추가하는 것이 좋은 상황은?
사실 이 부분은 정답이 있다기 보다는, 본인의 비즈니스 환경에 맞는 답을 찾아야 한다.

실무 관점에서 **한 명의 회원이 갖는 주문 건수가 평균적으로 수천 건을 넘어서고, 해당 조건으로 조회하는 기능이 서비스의 핵심적인 성능 병목점**이 될 때 `(member_id, order_status, ordered_at)` 복합 인덱스 추가를 고려해볼 수 있다.

하지만 단순히 데이터 건수만으로 결정하기보다는, 여러 요소를 종합적으로 판단하는 것이 중요하다.
예를 들어서 주문 내역에 데이터가 많지 않아도 메인 화면에 주문 내역을 항상 노출해야 한다면, 그래서 주문 내역을 조회하는 기능이 매우 빈번하게 발생한다면 복합 인덱스 추가를 고려할 수 있다.

인덱스 추가를 결정하는 실무적 기준

1. 데이터 분포와 조회 효율성
가장 중요한 기준이다. `member_id`로 데이터를 필터링했을 때 남는 데이터의 양이 얼마나 되는지가 핵심이다.

- **수백 건 이하**: 한 회원의 주문이 수십 ~ 수백 건 수준이라면 `member_id` 인덱스만으로 충분히 빠르다. 이 정도 데이터는 메모리에서 필터링하는 속도가 매우 빨라서 사용자가 성능 저하를 거의 체감할 수 없다. 추가 인덱스로 인한 쓰기 성능 저하와 관리 비용이 더 클 수 있다.
- **수천 건 이상**: B2B 서비스나 특수 유통몰처럼 한 명의 사용자(예: 사업자 회원)가 수천, 수만 건의 주문을 생성하

는 경우가 있다. 이 경우 `member_id`로만 필터링해도 여전히 많은 데이터가 남게 되므로, `order_status`와 `ordered_at`로 추가 필터링하는 과정에서 성능 저하가 발생할 수 있다. 이럴 때 복합 인덱스가 효과를 발휘하기 시작한다.

- **서비스 규모가 아주 큰 경우**: 전 국민이 사용하는 규모의 큰 쇼핑몰 서비스라면 데이터도 많고 트래픽도 많다. 이런 경우 약간의 지연도 시스템에 민감하게 반응하기 때문에 인덱스를 고려해야 한다.

2. 쓰기(Write) 작업의 빈도와 중요도

쇼핑몰의 `orders` 테이블은 `SELECT` 만큼이나 `INSERT`와 `UPDATE`가 매우 빈번하게 일어나는 곳이다.

- 주문 생성(`INSERT`)은 계속 발생한다.
- 주문 상태(`order_status`) 변경(`UPDATE`)은 '주문 완료' ➜ '결제 확인' ➜ '배송 준비' ➜ '배송 중' ➜ '배송 완료' 등으로 계속해서 일어난다.

복합 인덱스를 추가하면 이 모든 `INSERT`와 `UPDATE` 작업에 부하가 걸린다. **전체적인 주문 처리 속도가 느려지는 것은 사용자 경험에 치명적일 수 있다.** 따라서 읽기(조회) 성능의 개선 효과가 쓰기 성능 저하로 인한 손실보다 확실히 클 때만 인덱스를 추가해야 한다.

3. 실제 쿼리 패턴과 성능 측정 (가장 중요)

결국 가장 확실한 방법은 실제 데이터를 기반으로 테스트하는 것이다.

- **느린 쿼리(Slow Query) 로그 분석**: 데이터베이스의 느린 쿼리 로그를 분석했을 때, 해당 사용자 조회 쿼리가 지속적으로 상위권에 등장하며 시스템에 부하를 주고 있는지 확인한다.
- `EXPLAIN` **실행 계획 분석**: `EXPLAIN`을 사용해 실제 쿼리가 `member_id` 인덱스를 사용한 후 얼마나 많은 행(rows)을 스캔하는지 확인한다. 여기서 스캔하는 행의 수가 너무 많다고 판단되면 인덱스 추가를 검토한다.
- **성능 테스트**: 개발 환경에 인덱스와 데이터를 추가하고, 추가하기 전과 후의 조회 성능 및 주문 생성/수정 성능을 직접 비교 측정한다.

핵심은 "예상 가능한 확실한 경우가 아니라면 미리 짐작해서 최적화하지 말고, 문제가 발생하기 전에 데이터를 기반으로 판단하고 개선한다"는 것이다.

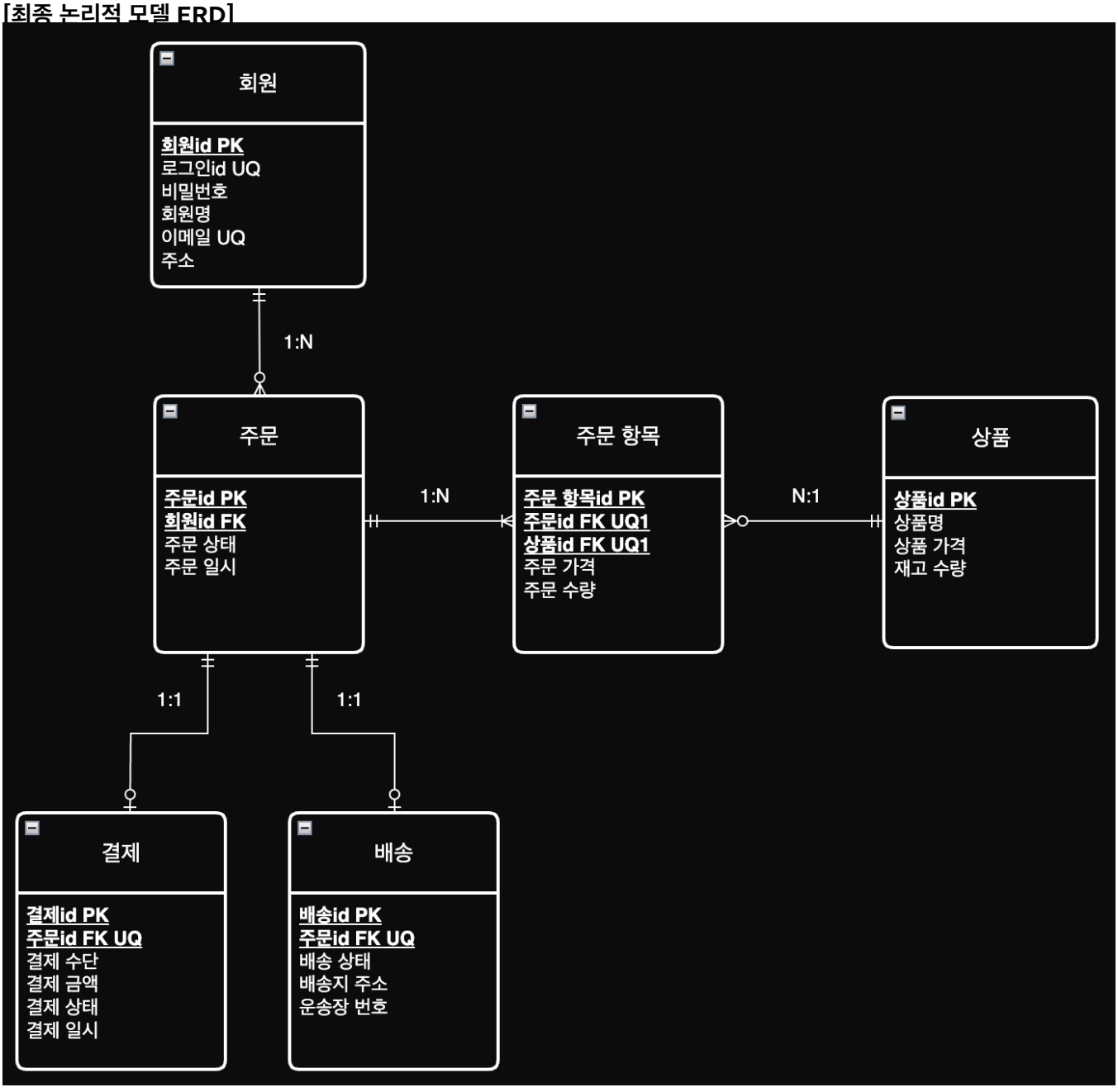
> **전투에 패한 장수는 용서할 수 있지만 경계에 실패한 장수는 용서할 수 없다.**
> 시스템에 장애는 언제든지 발생할 수 있다. 하지만 그 장애가 발생하기 전까지 아무런 징후를 파악하지 못하고 속수무책으로 당하는 것은 개발자의 명백한 책임이다. 인덱스 최적화도 마찬가지다. 무작정 인덱스를 만드는 것이 아니라, 느린 쿼리 로그를 통해 시스템의 병목을 '경계'하고, 데이터를 기반으로 '전투(최적화)'에 임해야 한다. 항상 시스템을 주의 깊게 관찰하고 문제가 발생했을 때 빠르게 대응할 수 있도록 준비하는 자세가 무엇보다 중요하다.

역정규화 - 실습

역정규화는 양날의 검과 같아서 신중하게 사용해야 한다. 우리는 어떤 상황에서, 왜 역정규화를 고려해야 하는지, 그리고 그로 인해 발생하는 트레이드오프는 무엇인지 명확하게 이해해야 한다. 참고로 역정규화는 성능 최적화를 위한 최후의 수단으로 고려해야 한다.

먼저 우리의 논리적 모델을 다시 한번 살펴보자.

[최종 논리적 모델 ERD]



이 모델은 정규화가 잘 되어 있어 데이터 중복이 없고, 일관성을 유지하기에 매우 좋은 구조다. 하지만 실제 운영 환경을 상상해 보자.

역정규화 - 중복 컬럼 추가

문제 상황

주문 내역 조회는 우리 쇼핑몰에서 가장 빈번하게 일어나는 핵심 기능 중 하나다. 사용자는 '마이페이지'에서 자신의 주문 내역을 볼 때, 각 주문에 어떤 상품들이 포함되어 있는지 상품명과 함께 확인하고 싶어 한다.

현재의 정규화된 모델에서 이 기능을 구현하려면, `order_item` 테이블과 `product` 테이블을 항상 `JOIN` 해야만 상품명(`product_name`)을 가져올 수 있다.

```
SELECT
    oi.order_id,
    p.product_name, -- 상품명을 위해 반드시 product 테이블을 JOIN 해야 한다.
    oi.order_price,
    oi.order_quantity
FROM
    order_item oi
JOIN
    product p ON oi.product_id = p.product_id
WHERE
    oi.order_id = 100;
```

주문 데이터가 수백만, 수천만 건으로 늘어난다면, 이 빈번한 `JOIN` 연산은 데이터베이스에 상당한 부하를 주게 되고, 주문 내역 페이지의 로딩 속도를 저하시키는 주범이 될 것이다.

"왜 우리는 매번 상품명을 알기 위해 `product` 테이블을 찾아가야만 할까? 주문 내역을 볼 때 거의 항상 상품명이 필요한데, 이 `JOIN` 비용을 없앨 수 있는 방법은 없을까?"

이 질문에 대한 해답이 바로 **역정규화**다. 가장 대표적인 역정규화 기법인 '중복 컬럼 추가'를 적용해 보자. 즉, `order_item` 테이블에 `product_name` 컬럼을 추가하여 상품명을 직접 저장하는 것이다.

단계적 설명

1단계: 컬럼 추가

먼저, `order_item` 테이블에 상품명을 저장할 `product_name` 컬럼을 추가한다.

```
ALTER TABLE order_item
ADD COLUMN product_name VARCHAR(100) NOT NULL COMMENT '주문 당시 상품명';
```

`COMMENT`를 추가하여 이 컬럼이 어떤 역할을 하는지 명확히 남겨두는 것은 좋은 습관이다.

> **참고**
> 테이블은 아직 만들지 않았지만, 이해를 돕기 위해 테이블이 만들어져 있다고 가정하겠다.

2단계: 데이터 저장 방식 변경

이제부터 새로운 주문이 들어와 `order_item` 데이터를 생성할 때, 애플리케이션 로직에서 `product` 테이블을 한 번 조회하여 상품명을 가져온 뒤, `order_item` 테이블의 `product_name` 컬럼에 함께 저장해 주어야 한다.

역정규화 적용 후

역정규화가 적용된 후, 주문 내역을 조회하는 쿼리는 어떻게 바뀔까?

```
-- 이제 product 테이블과의 JOIN이 필요 없다!
SELECT
    oi.order_id,
    oi.product_name, -- order_item 테이블에서 바로 상품명을 조회한다.
    oi.order_price,
    oi.order_quantity
FROM
    order_item oi
WHERE
    oi.order_id = 100;
```

`JOIN`이 사라진 매우 단순하고 빠른 쿼리가 완성되었다. 조회 성능 향상을 기대할 수 있다.

실무 경험 기반 설명

이렇게 하면 데이터 중복이 생기고, 만약 `product` 테이블에서 상품명이 바뀌면 `order_item`에 있는 상품명은 옛

날 이름 그대로 남아서 데이터가 달라지는 문제가 생길 수 있다.

이것이 바로 역정규화가 초래하는 **데이터 불일치(Inconsistency)의 위험**이며, 우리가 감수해야 할 트레이드오프다. 데이터 불일치 문제를 해결하려면, 상품명이 바뀔 때 마다 `order_item`에 저장된 모든 상품명도 함께 바꾸어야 한다.

사실 이것은 역정규화가 아니다.
사실 이것은 엄밀히 말해서 역정규화라기 보다는 앞서 놓친 비즈니스 요구사항을 물리적 구현 단계에서 발견한 것에 가깝다.

'주문' 데이터의 특성을 깊이 생각해 보자. 만약 고객이 'A 노트북'을 150만원에 주문했는데, 며칠 뒤 쇼핑몰에서 상품명을 'A 게이밍 노트북'으로 바꾸고 가격을 160만원으로 올렸다고 가정하자. 이때 고객의 과거 주문 내역까지 'A 게이밍 노트북'을 150만원에 산 것으로 바뀌어야 할까? 절대 아니다.

고객의 주문 내역은 주문이 일어난 **그 시점의 정보**를 그대로 보존해야 하는 **'역사적 데이터(Historical Data)' 또는 '스냅샷(Snapshot)'**이다. 즉, 주문 당시의 상품명(`product_name`)과 주문 당시의 가격(`order_price`)은 나중에 원본 상품 정보가 바뀌더라도 절대 함께 변경되어서는 안 된다.

이런 비즈니스 규칙 덕분에, `order_item`에 `product_name`을 복사해두는 것은 데이터 불일치 문제가 아니라, 오히려 비즈니스 요구사항을 더 정확하게 만족시키는 훌륭한 설계가 된다. 우리는 이미 `order_price`를 통해 이 원칙을 적용하고 있었다. `product_name`도 마찬가지다.

결론적으로, 이 경우에는 **성능 향상**과 **비즈니스 요구사항 만족**이라는 두 마리 토끼를 모두 잡을 수 있었다.

역정규화 - 파생 컬럼 추가(계산된 값 저장)

문제 상황
쇼핑몰 관리자는 주문 관리 페이지에서 각 주문의 '총 결제 금액'을 한눈에 파악하고 싶어 한다. 이 총 금액을 기준으로 매출 통계를 내거나, 특정 금액 이상 주문한 고객을 필터링하는 기능이 필요하다. 고객의 경우도 마찬가지다. 자신의 주문 관리 페이지에서 각 주문의 '총 결제 금액'을 한눈에 파악하고 싶어 한다.

현재 모델에서 주문 ID가 `100`번인 주문의 총 결제 금액을 계산하려면, `order_item` 테이블에서 해당 주문에 속한 모든 상품들의 `order_price`와 `order_quantity`를 곱한 값을 다시 모두 더하는 복잡한 집계 쿼리를 실행해야 한다.

```
SELECT
```

```
    o.order_id,
    o.ordered_at,
    SUM(oi.order_price * oi.order_quantity) AS total_amount -- 매번 복잡한 계산이
필요하다.
FROM
    orders o
JOIN
    order_item oi ON o.order_id = oi.order_id
WHERE
    o.order_id = 100
GROUP BY
    o.order_id, o.ordered_at;
```

주문 목록을 보여줄 때마다 모든 주문에 대해 이 계산을 반복한다면, 데이터베이스는 엄청난 계산 부하를 견뎌야 한다.

"왜 우리는 총 주문 금액을 알기 위해 매번 이렇게 복잡한 계산을 반복해야 할까? 어차피 한번 계산된 총 금액은 잘 변하지 않는데, 이 계산 결과를 미리 저장해두면 안 될까?"

이것이 바로 '파생 컬럼 추가' 또는 '계산된 값 저장'이라는 역정규화 기법이다. `orders` 테이블에 `total_amount` 라는 컬럼을 추가하여, 주문이 생성되는 시점에 총 주문 금액을 미리 계산해서 저장해두는 것이다.

단계적 설명

1단계: 컬럼 추가

`orders` 테이블에 총 주문 금액을 저장할 `total_amount` 컬럼을 추가한다.

```
ALTER TABLE orders
ADD COLUMN total_amount INT NOT NULL COMMENT '총 주문 금액';
```

2단계: 데이터 저장 방식 변경

이제 애플리케이션에서 주문을 생성하는 로직이 조금 더 복잡해진다.

1. `order_item`에 각 상품 정보를 저장한다.
2. 저장된 `order_item`들을 바탕으로 총 주문 금액을 계산한다.
3. 계산된 총 금액을 `orders` 테이블의 `total_amount` 컬럼에 저장한다.

3단계: 조회 쿼리 개선

이제 총 주문 금액을 조회하는 쿼리는 `JOIN`과 `SUM`이 모두 사라진, 매우 빠르고 단순한 쿼리로 바뀐다.

```
-- 계산 없이 orders 테이블에서 바로 총 금액을 조회한다.
SELECT
    order_id,
    ordered_at,
    total_amount
FROM
    orders
WHERE
    order_id = 100;
```

실무 경험 기반 설명

이 기법 역시 트레이드오프가 명확하다. 우리는 **읽기(SELECT) 성능을 극적으로 향상**시키는 대신, **쓰기(INSERT/UPDATE) 로직의 복잡성과 부하를 증가**시켰다.

또한 데이터 정합성을 유지하기 위한 책임이 애플리케이션 개발자에게 넘어왔다. 만약 주문 생성 후 `total_amount`를 업데이트하는 로직을 빠뜨리거나, 주문 상품이 일부 취소되었을 때 `total_amount`를 다시 계산하거나, 차감하는 로직을 제대로 구현하지 않으면, 실제 주문 내역과 총 금액이 맞지 않는 심각한 데이터 불일치 문제가 발생한다.

따라서 이 기법은 **조회는 매우 빈번하지만, 데이터 변경은 상대적으로 드물고, 쓰기 시점의 약간의 부하를 감수할 수 있는 경우**에 매우 효과적이다. 쇼핑몰의 주문 데이터가 바로 이런 특성에 해당한다.

역정규화 - 테이블 통합

문제 상황

어떤 개발자가 이런 제안을 했다고 가정해 보자.

"가만 보니 `orders`와 `delivery`는 거의 항상 1:1 관계입니다. 주문 정보를 볼 때 배송 상태나 주소도 같이 보는 경우가 많으니, 두 테이블을 하나로 합쳐서 `JOIN`을 아예 없애버리면 성능에 더 좋지 않을까요?"

매우 그럴듯하게 들리는 주장이다. `delivery` 테이블의 컬럼들(`ship_addr`, `delivery_status`, `tracking_no`)을 `orders` 테이블로 옮기면, `JOIN`이 하나 줄어드니 조회 성능이 빨라질 것이다. 그리고 관리해야 하는 테이블도 하나 줄어들어서 개발도 더 편리할 것 같다.

하지만 이 방법은 매우 신중하게 고민해야 한다.
"왜 우리는 이처럼 명확해 보이는 성능상의 이점에도 불구하고, 테이블 통합을 신중하게 다시 생각해봐야 할까?"

데이터베이스 전문가라면 이 제안이 **좋은 방법이 아니라고** 단호하게 말할 것이다. 그 이유는 `JOIN`을 한번 줄여서 얻는 사소한 이점보다, 테이블을 분리함으로써 얻는 구조적인 유연성과 데이터 관리의 명확성이 훨씬 더 중요하기 때문이다. **역정규화는 항상 트레이드오프를 고려해야 하며**, 이 경우는 얻는 것 보다 잃는 것이 훨씬 크다.

데이터베이스 전문가의 반대 이유

1. 데이터의 생명주기(Lifecycle)가 다르다

- **주문(`orders`)** 데이터는 고객이 '주문하기' 버튼을 누르는 즉시 생성된다. 이때 주문 상품, 금액 등은 모두 확정된다.
- **배송(`delivery`)** 데이터는 주문 이후에 생성되고, 상태가 계속 변한다. 운송장 번호(`tracking_no`)는 상품이 실제 발송 준비가 되어야만 입력될 수 있다.
- 만약 두 테이블을 합치면, 주문이 막 생성된 시점에는 `tracking_no` 같은 배송 관련 컬럼들이 모두 `NULL` 값으로 채워져 있게 된다. 불필요한 `NULL` 값은 저장 공간을 낭비하고, 데이터의 의미를 불분명하게 만든다.

2. 데이터 변경 빈도가 다르다

- `orders`나 `order_item`의 핵심 데이터(주문자, 주문 상품, 가격)는 한번 생성되면 거의 변경되지 않는다.
- 반면 `delivery`의 `delivery_status` 컬럼은 '준비중' ➔ '배송중' ➔ '배송완료' 처럼 상태가 매우 빈번하게 `UPDATE` 된다.
- 만약 하나의 거대한 테이블로 합쳐져 있다면, 단지 배송 상태 하나를 바꾸기 위해 주문 정보까지 포함된 훨씬 큰 데이터를 건드려야 한다. 이는 불필요한 I/O를 유발하고, 데이터베이스 락(Lock) 경합의 가능성을 높여 오히려 전체적인 시스템 성능을 저하시킬 수 있다.

3. 테이블의 책임과 응집도가 깨진다 (SRP 위반)

- `orders` 테이블의 책임은 **'누가, 언제, 무엇을 주문했는가'**에 대한 정보를 관리하는 것이다.
- `delivery` 테이블의 책임은 **'그 주문이 어떻게, 어디로, 어떤 상태로 배송되고 있는가'**를 관리하는 것이다.
- 두 테이블을 합치는 것은, 성격이 다른 두 책임(주문, 배송)을 하나의 테이블에 억지로 구겨 넣는 행위다. 이는 소프트웨어 설계의 기본 원칙인 **단일 책임 원칙(Single Responsibility Principle)**을 위배하며, 테이블의 의미를 모호하게 만들고 향후 유지보수를 어렵게 만든다.

4. 확장성을 심각하게 저해한다

- 지금은 주문 하나당 배송 하나(1:1)지만, 서비스가 성장하면 어떻게 될까?
- '묶음 배송'이나 '분할 배송'(하나의 주문에 포함된 여러 상품을 여러 곳으로 나누어 보내는 경우) 기능이 추가된다면, `orders`와 `delivery`의 관계는 1:N으로 바뀌어야 한다.

- 테이블이 분리되어 있다면, 이런 요구사항 변경에 유연하게 대처할 수 있다. 하지만 이미 하나의 테이블로 통합되어 있다면, 데이터베이스 구조 전체를 뜯어고쳐야 하는 대공사가 필요하게 된다.

5. 선택적 관계(Optional Relationship)에 대한 처리

- 만약 우리 쇼핑몰에서 E-Book이나 온라인 강의 같은 디지털 상품을 판매하게 된다면 어떻게 될까?
- 디지털 상품은 배송 자체가 필요 없다. 통합된 테이블에서는 이런 주문이 들어올 때마다 모든 배송 관련 컬럼이 `NULL`로 남게 된다. 이는 데이터 무결성을 해치고, `delivery_status`가 `NULL`인 데이터가 '배송 전'인지 '배송 불가 상품'인지 구분하기 어렵게 만든다.

6. 성능 이점이 크지 않음

- `orders`와 `delivery`의 JOIN은 `order_id`라는 PK/FK/UNIQUE 키를 기반으로 한 1:1 조인이다. 이런 종류의 JOIN은 인덱스가 잘 설정되어 있다면 데이터베이스가 매우 효율적으로 처리하여 성능 부하가 크지 않다. 즉, 테이블을 합쳐서 얻는 성능상 이점은 매우 미미한데, 위에서 언급한 구조적 문제점들은 매우 크다.

결과적으로 `delivery`는 그대로 분리한다

이러한 트레이드오프를 신중하게 고려한 결과, `delivery` 테이블을 `orders` 테이블로 통합하는 것은 장기적으로 더 큰 문제를 야기할 수 있는, 좋지 않은 선택이라는 결론에 도달했다.

`JOIN`**을 한 번 줄이는 단기적인 성능 이익**보다, **데이터 모델의 명확성, 유연성, 확장성을 지키는 장기적인 이익**이 훨씬 더 크다. 따라서 우리의 최종 물리적 모델에서도 `delivery` 테이블은 `orders`와 분리된 상태를 그대로 유지한다.

실무 이야기

이번에 다룬 `orders`와 `delivery` 테이블의 통합 여부 문제는 비단 데이터베이스 설계에만 국한된 이야기가 아니다. 이것은 객체 지향 프로그래밍에서 클래스를 어떻게 나눌 것인가와 같은, 소프트웨어 설계 전반을 관통하는 매우 근본적인 고민이다. 무언가를 분리할지, 아니면 합칠지를 결정해야 할 때, 오늘 우리가 배운 두 가지 기준인 **'데이터의 생명주기'**와 **'변경 빈도'**를 떠올리면 도움이 될 것이다.

- **생명주기가 다른 데이터는 분리하라:** 주문(`orders`)은 결제 시점에 '생성'되고 거의 변하지 않는다. 반면 배송(`delivery`)은 주문 이후에 '생성'되고, 상태가 계속 '변경'되다가 배송 완료 시점에 사실상 생명이 다한다. 이렇게 태어나고 죽는 시점, 그리고 살아가는 모습이 다른 데이터들을 하나의 테이블에 묶어두면 앞서 본 것처럼 수많은 `NULL`값과 불필요한 복잡성을 낳게 된다. 각자의 생명주기에 맞는 집(테이블)을 만들어 주는 것이 가장 자연스러운 설계다.

- **변경 빈도가 다른 데이터는 분리하라:** 주문의 핵심 정보는 한번 정해지면 바뀔 일이 거의 없는 정적인 데이터다. 반면 배송 상태는 계속해서 `UPDATE`가 일어나는 동적인 데이터다. 만약 이 둘을 합치면, 우리는 고작 배송 상태 하나를 바꾸기 위해 거의 바뀌지 않는 주문 정보까지 매번 함께 조회하고 수정해야 하는 비효율을 감수해야 한다.

이는 데이터베이스의 `UPDATE` 성능 저하와 락(Lock) 경합의 원인이 되기도 한다. 자주 바뀌는 데이터는 따로 분리해서, 다른 데이터에 미치는 영향을 최소화하는 것이 현명하다.

결국 좋은 설계는 **'함께 변경될 가능성이 높은 것들은 모으고, 그렇지 않은 것들은 분리하는 것'**에서 출발한다. 이 원칙 하나만 잘 기억하고 적용해도, 훨씬 더 유연하고 확장성 있는 데이터베이스 모델을 만들 수 있을 것이다.

쇼핑몰 테이블 정의서

테이블 정의서

이제 테이블 정의서를 완성해보자.

강의에서는 표의 오른쪽 지면 부족으로 다음과 같이 축약해서 정리했다.

- 기본값을 비고에 포함했다. 기본값은 별도의 항목으로 설명하는 것이 좋다.
- NULL을 제약(제약 조건)에 포함했다. `NULL`은 별도의 항목으로 설명하는 것이 좋다.
  - 지면상 `NOT NULL`은 표현하지 않겠다. (여기서 기본은 NOT NULL로 보면 된다.)
- `AUTO_INCREMENT` 기본값은 생략했다.
- `created_at`, `updated_at`의 기본값은 생략했다.
- Unique ➜ UQ

한글명을 영문명으로 변환할 때 앞서 준비한 **용어 사전**을 사용하자.

member

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

No.	컬럼 한글명	컬럼 영문명	데이터 타입	제약 조건	비고
1	회원 ID	`member_id`	`BIGINT`	PK	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 암호화하여 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`		회원 가입 시점
8	수정일	`updated_at`	`DATETIME`		데이터 수정 시 자동 갱신

product

테이블 한글명	**상품**
테이블 영문명	`product`
테이블 설명	쇼핑몰에서 판매하는 상품의 정보를 저장하는 테이블

No.	**컬럼 한글명**	**컬럼 영문명**	**데이터 타입**	**제약 조건**	**비고**
1	상품 ID	`product_id`	`BIGINT`	PK	상품의 고유 식별자 (대리 키)
2	상품명	`product_name`	`VARCHAR(100)`		상품의 이름
3	상품 가격	`product_price`	`INT`		상품의 현재 판매 가격
4	재고 수량	`stock_quantity`	`INT`		상품의 현재 재고 수량 기본값: `0`

5	상품 등록일	`created_at`	`DATETIME`		상품이 시스템에 등록된 시점
6	수정일	`updated_at`	`DATETIME`		데이터 수정 시 자동 갱신

- **인덱스**

인덱스명	컬럼	비고
`idx_product_name`	`product_name`	상품명 검색 성능 향상을 위한 인덱스

orders

테이블 한글명	주문
테이블 영문명	`orders`
테이블 설명	회원의 주문 정보를 저장하는 테이블

No.	컬럼 한글명	컬럼 영문명	데이터 타입	제약 조건	비고
1	주문 ID	`order_id`	`BIGINT`	PK	주문의 고유 식별자 (대리 키)
2	회원 ID	`member_id`	`BIGINT`	FK	주문한 회원 (`member.member_id` 참조)
3	주문일시	`ordered_at`	`DATETIME`		고객이 주문한 비즈니스 시점의 시간, 백엔드 애플리케이션에서 전달
4	주문 상태	`order_status`	`VARCHAR(20)`		예: ORDERED, CANCELED 등 기본값: `ORDERED`
5	총 주문 금액	`total_amount`	`INT`		**[역정규화]** 주문 상품 금액의 총합

6	생성일	created_at	DATETIME		주문 데이터가 시스템에 생성된 시점
7	수정일	updated_at	DATETIME		주문 상태 변경 등 데이터 수정 시 자동 갱신

- **인덱스**

인덱스명	컬럼	비고
idx_order_status_ordered_at	order_status, ordered_at	관리자의 주문 상태 및 기간별 조회 성능 향상을 위한 복합 인덱스

order_item

테이블 한글명	주문 항목
테이블 영문명	order_item
테이블 설명	특정 주문에 포함된 개별 상품들의 상세 내역을 저장하는 테이블

No.	컬럼 한글명	컬럼 영문명	데이터 타입	제약 조건	비고
1	주문 상품 ID	order_item_id	BIGINT	PK	주문 상품의 고유 식별자 (대리 키)
2	주문 ID	order_id	BIGINT	FK	해당 상품이 속한 주문 (orders.order_id 참조)
3	상품 ID	product_id	BIGINT	FK	주문된 상품 (product.product_id 참조)
4	주문 상품명	product_name	VARCHAR(100)		**[역정규화]** 주문 당시의 상품명 (스냅샷)

5	주문 가격	`order_price`	`INT`		주문 당시의 개별 상품 가격 (스냅샷)
6	주문 수량	`order_quantity`	`INT`		주문한 상품의 개수
7	생성일	`created_at`	`DATETIME`		데이터가 시스템에 생성된 시점
8	수정일	`updated_at`	`DATETIME`		데이터 수정 시 자동 갱신

- **인덱스**

인덱스명	컬럼	비고
`uq_order_id_product_id`	`order_id`, `product_id`	하나의 주문에 동일한 상품을 담을 수 없는 유니크 제약 조건

delivery

테이블 한글명	배송
테이블 영문명	`delivery`
테이블 설명	주문에 대한 배송 정보를 저장하는 테이블 (주문과 1:1 관계)

No.	컬럼 한글명	컬럼 영문명	데이터 타입	제약 조건	비고
1	배송 ID	`delivery_id`	`BIGINT`	PK	배송의 고유 식별자 (대리 키)
2	주문 ID	`order_id`	`BIGINT`	FK, UQ	배송될 주문 (`orders.order_id` 참조)

3	배송 상태	`delivery_status`	`VARCHAR(20)`		예: READY, SHIPPING, COMPLETED 등 기본값: `'READY'`
4	운송장 번호	`tracking_no`	`VARCHAR(50)`	NULL	배송 시작 시 택배사에서 발급하는 번호
5	배송지	`ship_addr`	`VARCHAR(255)`		상품이 실제 배송될 주소
6	생성일	`created_at`	`DATETIME`		배송 데이터가 시스템에 생성된 시점
7	수정일	`updated_at`	`DATETIME`		배송 상태 변경 등 데이터 수정 시 자동 갱신

pay

테이블 한글명	결제
테이블 영문명	`pay`
테이블 설명	주문에 대한 결제 정보를 저장하는 테이블 (주문과 1:1 관계)

No.	컬럼 한글명	컬럼 영문명	데이터 타입	제약 조건	비고
1	결제 ID	`pay_id`	`BIGINT`	PK	결제의 고유 식별자 (대리키)
2	주문 ID	`order_id`	`BIGINT`	FK, UQ	결제 대상 주문 (`orders.order_id` 참조)
3	결제 수단	`pay_method`	`VARCHAR(50)`		예: CREDIT_CARD, BANK_TRANSFER 등

4	결제 금액	pay_amount	INT		실제 은행, 카드 등의 결제가 이루어진 금액, 포인트 할인 등의 금액은 제외된다.
5	결제 상태	pay_status	VARCHAR(20)		예: PAID, FAILED, CANCELED 등
6	결제일시	paid_at	DATETIME	NULL	결제가 최종 완료된 시점, 비즈니스 관점에서 시간, 은행, 신용카드 사에서 결제된 시간
7	생성일	created_at	DATETIME		결제 데이터가 시스템에 생성된 시점
8	수정일	updated_at	DATETIME		결제 상태 변경 등 데이터 수정 시 자동 갱신

생성일과 수정일

실무에서는 모든 테이블에 **생성일(`created_at`)**과 **수정일(`updated_at`)** 컬럼을 넣는 것이 거의 표준처럼 여겨진다. 이 두 컬럼은 데이터가 언제 생성되고 마지막으로 수정되었는지 기록하는, 아주 중요한 '감사(Audit)' 정보를 담고 있다.

- `created_at`: 데이터가 처음 생성된 시점을 기록한다. 이 값은 한 번 정해지면 절대 변하지 않는다.
- `updated_at`: 데이터가 마지막으로 수정된 시점을 기록한다. 행의 어떤 컬럼이라도 변경되면 이 필드의 값은 현재 시간으로 자동 업데이트된다.

이 정보가 있으면 데이터 관련 문제가 발생했을 때 변경 이력을 추적하여 원인을 파악하기가 매우 용이해진다. 예를 들어, "어떤 회원의 주소가 어젯밤에 이상하게 바뀌었어요!"라는 문의가 들어왔을 때, `updated_at`을 보면 언제 변경이 일어났는지 즉시 알 수 있고, 해당 시간대의 로그를 집중적으로 분석하여 원인을 찾을 수 있다.

앞서 학습한 것 처럼 MySQL의 자동 업데이트 기능을 사용하자.

> ✨ **실무 팁 - 등록자(created_by), 수정자(updated_by)**

> 실무에서는 등록일(`created_at`), 수정일(`updated_at`)은 물론이고
> 등록자(`created_by`), 수정자(`updated_by`)도 추가해서 관리하는 것이 좋다.
> 이렇게 하면 문제가 발생했을 때 관리자가 이 데이터를 수정했는지, 아니면 사용자가 이 데이터를 수정했는지 명확하게 인지할 수 있다.
> 등록자, 수정자는 데이터베이스에서 자동화할 수는 없고, 대신에 애플리케이션에서 관리해야 한다.
> 스프링 데이터 JPA와 같은 기술은 애플리케이션에서 이런 부분을 자동화해서 관리할 수 있다.

주문일시는 왜 따로 있을까? (`created_at` vs. `ordered_at`)

`orders` 테이블 정의서를 보면 `created_at`(생성일)과 `ordered_at`(주문일시)이라는, 언뜻 보기에 비슷해 보이는 두 개의 시간 관련 컬럼이 있는 것을 보고 의문을 가졌을 것이다. "데이터가 생성된 시간이 주문 시간 아닌가?"라고 생각하기 쉽다. 하지만 실무에서는 이 둘을 명확히 구분하는 것이 매우 중요하다. 결론부터 말하자면, `created_at`은 **시스템의 시간**이고, `ordered_at`은 **비즈니스의 시간**을 나타낸다.

시스템의 시간 vs. 비즈니스의 시간

- `created_at` **(시스템의 시간)**: 이 컬럼은 데이터베이스에 **실제** `INSERT` **쿼리가 실행되어 행(Row)이 생성된 시점**을 기록한다. 이것은 순수하게 기술적인 정보다. 데이터가 언제 물리적으로 저장되었는지를 나타내며, 주로 시스템 감사(Audit), 로그 추적, 데이터 백업 및 복구 시점 확인 등 시스템 관리 목적으로 사용된다. 이 값은 데이터베이스가 `DEFAULT CURRENT_TIMESTAMP`와 같은 기능을 통해 자동으로 채우도록 설정하는 것이 일반적이다.

- `ordered_at` **(비즈니스의 시간)**: 이 컬럼은 **비즈니스 이벤트가 발생한 시점**, 즉 고객이 우리 쇼핑몰에서 '주문하기' 버튼을 누른 바로 그 순간을 기록한다. 이것은 고객에게 보여주고, 비즈니스 로직을 처리하는 데 사용되는 매우 중요한 정보다. 예를 들어 "오후 3시 이전 주문 건은 당일 배송"과 같은 정책은 바로 이 `ordered_at`을 기준으로 판단해야 한다. 이 값은 사용자의 요청을 받은 백엔드 애플리케이션이 생성하여 데이터베이스에 전달해야 한다.

두 시간이 달라지는 실무 시나리오

"그래도 두 시간은 거의 같지 않나요?"라고 생각할 수 있다. 대부분의 경우에는 그 차이가 밀리초(ms) 단위로 매우 작을 것이다. 하지만 시스템의 안정성과 데이터의 정확성을 보장해야 하는 실무에서는 두 시간이 의미 있게 달라지는 경우가 분명히 존재한다.

시나리오 1: 시스템 장애 또는 지연

가장 흔한 경우다. 사용자가 많아 서버에 부하가 걸리거나, 데이터베이스에 일시적인 성능 저하가 발생했다고 가정해 보자.

1. **고객 A**가 `14:59:59`에 '주문하기' 버튼을 클릭했다. (비즈니스 이벤트 발생)
2. 백엔드 애플리케이션은 이 요청을 받고, 현재 시간 `14:59:59`를 `ordered_at` 값으로 확정한다.
3. 애플리케이션이 주문 데이터를 `orders` 테이블에 `INSERT` 하려고 시도한다.
4. 하지만 데이터베이스에 부하가 몰려있어 쿼리가 즉시 실행되지 못하고 2초간 대기했다.
5. `INSERT` 쿼리가 `15:00:01`에 최종적으로 실행되어 데이터가 저장되었다.

이 경우, 테이블에는 다음과 같이 기록된다.

- `ordered_at`: `2025-08-27 14:59:59`
- `created_at`: `2025-08-27 15:00:01`

만약 `created_at`만 있었다면 이 주문은 '오후 3시 이후 주문'으로 잘못 처리되어 당일 배송에서 누락되었을 것이다. `ordered_at`을 별도로 관리했기 때문에 우리는 고객의 주문 시점을 정확히 기록하고 비즈니스 규칙을 올바르게 적용할 수 있다.

시나리오 2: 데이터 마이그레이션
기존에 운영하던 쇼핑몰의 데이터를 새로운 시스템으로 이전하는 상황을 생각해 보자. 작년(2024년)의 주문 데이터를 옮겨야 한다.

1. 2024년 12월 25일에 발생했던 주문 데이터를 새로운 `orders` 테이블에 `INSERT` 한다.
2. 이때, `ordered_at` 값은 당연히 원래 주문이 발생했던 `2024-12-25 10:30:00`으로 지정해서 넣어야 한다.
3. 하지만 `created_at` 값은 이 데이터가 새로운 시스템에 생성된 시점인 `2025-08-27 18:00:00` (마이그레이션 작업 시간)으로 기록될 것이다.

이처럼 과거 데이터를 이전하거나 복원할 때, 비즈니스 발생 시점과 데이터 생성 시점은 완전히 달라진다. 두 컬럼을 분리하지 않으면 과거 데이터의 중요한 시간 정보를 잃게 된다.

이처럼 `created_at`**과 같은 시스템 컬럼은 기술적인 무결성을 위해,** `ordered_at`**과 같은 비즈니스 컬럼은 비즈니스 로직의 정확성을 위해 반드시 분리하여 관리**해야 한다.
`pay` 테이블의 결제 일시(`paid_at`)도 비즈니스 시간이다. 결제가 최종 완료된 시점을 의미하는데, 여기서는 은행, 신용카드 사에서 결제된 시간을 의미한다. 이 시간을 확인해야 결제에 문제가 발생했을 때 은행, 카드사와 정확한 시간을 맞추어 확인할 수 있다.

쇼핑몰 DDL과 DB 만들기

지금까지 논의한 역정규화 전략과 감사 컬럼 추가를 모두 반영하여, 우리 쇼핑몰 데이터베이스의 최종 DDL을 완성해 보자.

`member` 테이블

```
CREATE TABLE member (
  member_id  BIGINT    NOT NULL AUTO_INCREMENT, -- 회원 ID (PK)
  login_id  VARCHAR(50)  NOT NULL,         -- 로그인 ID
  password  VARCHAR(255) NOT NULL,         -- 비밀번호 (암호화)
  member_name    VARCHAR(50)  NOT NULL,         -- 이름
  email    VARCHAR(100) NOT NULL,         -- 이메일
  addr   VARCHAR(255) NULL,           -- 주소
  created_at DATETIME   NOT NULL DEFAULT CURRENT_TIMESTAMP,
  updated_at DATETIME   NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,

  PRIMARY KEY (member_id),
  UNIQUE KEY uq_login_id (login_id),
  UNIQUE KEY uq_email (email)
);
```

`product` 테이블

```
CREATE TABLE product (
  product_id  BIGINT    NOT NULL AUTO_INCREMENT, -- 상품 ID (PK)
  product_name     VARCHAR(100) NOT NULL,         -- 상품명
  product_price     INT       NOT NULL,         -- 가격
  stock_quantity INT       NOT NULL DEFAULT 0,    -- 재고 수량
  created_at  DATETIME   NOT NULL DEFAULT CURRENT_TIMESTAMP,
  updated_at  DATETIME   NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,

  PRIMARY KEY (product_id),
  INDEX idx_product_name (product_name) -- 상품명 검색용 인덱스
);
```

- `idx_product_name` 인덱스를 추가했다.

`orders` 테이블 (역정규화, 인덱스 적용)

```
CREATE TABLE orders (
  order_id  BIGINT    NOT NULL AUTO_INCREMENT, -- 주문 ID (PK)
  member_id  BIGINT    NOT NULL,         -- 회원 ID (FK)
  ordered_at DATETIME   NOT NULL,     -- 주문일 (애플리케이션에서 생성)
  order_status   VARCHAR(20)  NOT NULL DEFAULT 'ORDERED', -- 주문 상태
  total_amount INT      NOT NULL,         -- 총 주문 금액 (역정규화)
  created_at DATETIME   NOT NULL DEFAULT CURRENT_TIMESTAMP,
  updated_at DATETIME   NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,

  PRIMARY KEY (order_id),
  CONSTRAINT fk_orders_member FOREIGN KEY (member_id)
    REFERENCES member (member_id),
  INDEX idx_order_status_ordered_at (order_status, ordered_at) -- 관리자용 주문 조
회 인덱스
);
```

- `total_amount` 파생 컬럼(역정규화)이 추가되었다. 이를 통해 매번 총액을 계산하는 복잡한 집계 쿼리를 피할 수 있다.
- `idx_order_status_ordered_at` 복합 인덱스가 추가되었다. 관리자가 주문 상태와 기간으로 주문을 조회할 때 성능을 향상시킨다.

`order_item` 테이블 (역정규화 적용)

```
CREATE TABLE order_item (
  order_item_id BIGINT    NOT NULL AUTO_INCREMENT, -- 주문 상품 ID (PK)
  order_id   BIGINT    NOT NULL,         -- 주문 ID (FK)
  product_id  BIGINT    NOT NULL,         -- 상품 ID (FK)
  product_name VARCHAR(100) NOT NULL,         -- 주문 당시 상품명 (역정규화)
  order_price INT      NOT NULL,         -- 주문 당시 가격
  order_quantity    INT      NOT NULL,         -- 주문 수량
  created_at  DATETIME   NOT NULL DEFAULT CURRENT_TIMESTAMP,
  updated_at  DATETIME   NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
```

```
  PRIMARY KEY (order_item_id),
  CONSTRAINT fk_order_item_orders FOREIGN KEY (order_id)
    REFERENCES orders (order_id),
  CONSTRAINT fk_order_item_product FOREIGN KEY (product_id)
    REFERENCES product (product_id),

  -- [추가됨] 주문 ID와 상품 ID에 대한 유니크 제약 조건
  CONSTRAINT uq_order_id_product_id UNIQUE (order_id, product_id)
);
```

- `product_name` 컬럼이 추가되었다. 주문 내역 조회 시 `product` 테이블과의 `JOIN` 없이 상품명을 바로 확인할 수 있다.

`delivery` 테이블 (통합하지 않고 유지)

```
CREATE TABLE delivery (
  delivery_id  BIGINT    NOT NULL AUTO_INCREMENT, -- 배송 ID (PK)
  order_id    BIGINT    NOT NULL,         -- 주문 ID (FK)
  delivery_status     VARCHAR(20)  NOT NULL DEFAULT 'READY',-- 배송 상태
  tracking_no VARCHAR(50)  NULL,           -- 운송장 번호
  ship_addr  VARCHAR(255) NOT NULL,         -- 배송지
  created_at   DATETIME   NOT NULL DEFAULT CURRENT_TIMESTAMP,
  updated_at   DATETIME   NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,

  PRIMARY KEY (delivery_id),
  UNIQUE KEY uq_delivery_order_id (order_id),
  CONSTRAINT fk_delivery_orders FOREIGN KEY (order_id)
    REFERENCES orders (order_id)
);
```

- `delivery` 테이블은 `orders`와 통합하지 않고 독립적으로 유지한다. 이를 통해 데이터 모델의 유연성과 확장성을 확보했다.

`pay` 테이블 생성 DDL

```sql
CREATE TABLE pay (
  pay_id  BIGINT    NOT NULL AUTO_INCREMENT, -- 결제 ID (PK)
  order_id   BIGINT    NOT NULL,         -- 주문 ID (FK, Unique)
  pay_method VARCHAR(50)  NOT NULL,         -- 결제 수단
  pay_amount    INT      NOT NULL,         -- 결제 금액
  pay_status    VARCHAR(20)  NOT NULL,         -- 결제 상태
  paid_at DATETIME   NULL,           -- 결제 완료일
  created_at  DATETIME   NOT NULL DEFAULT CURRENT_TIMESTAMP,
  updated_at  DATETIME   NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,

  PRIMARY KEY (pay_id),
  UNIQUE KEY uq_pay_order_id (order_id), -- 주문 하나당 결제는 하나만 존재해야 함 (1:1
관계 보장)
  CONSTRAINT fk_pay_orders FOREIGN KEY (order_id)
    REFERENCES orders (order_id)
);
```

이로써 우리는 정규화된 논리적 모델을 기반으로, 실제 운영 환경의 성능을 고려한 전략적인 역정규화, 그리고 데이터의 변경 이력을 추적하기 위한 감사 컬럼까지 모두 적용하여 최종 물리적 모델을 완성했다. 이제 이 DDL을 사용해 실제 데이터베이스에 테이블을 생성하고 우리 쇼핑몰 서비스를 만들어나갈 준비를 마친 것이다.

쇼핑몰 DB 만들기

이제 최종적으로 완성된 DDL과 샘플 데이터까지 포함하여, 우리의 '쇼핑몰' 데이터베이스를 처음부터 끝까지 완전히 구축할 수 있는 통합 스크립트를 만들어 보자. 이 스크립트 하나만 있으면, 언제 어디서든 동일한 구조와 데이터를 가진 데이터베이스를 즉시 생성할 수 있다.

스크립트는 다음과 같은 순서로 구성된다.

1. **데이터베이스 초기화**: 만약 기존에 `my_shop3_ex` 데이터베이스가 있다면 삭제하고, 깨끗한 상태에서 새로 만든다.
2. **테이블 초기화**: 생성할 테이블들이 이미 존재한다면 먼저 삭제한다. 외래 키(FK) 종속성을 고려하여 자식 테이블부터 삭제해야 한다.
3. **테이블 생성 (DDL)**: `member`, `product`, `orders`, `order_item`, `delivery`, `pay` 순서로 테이블을 생성한다.
4. **샘플 데이터 입력 (DML)**: 생성된 테이블에 실제 운영 상황과 유사한 샘플 데이터를 입력하여, 이후 개발 및 테스

트에 활용할 수 있도록 한다.

쇼핑몰 DB 구축 통합 스크립트 - DDL

```
-----------------------------------------
-- 데이터베이스 초기화
-----------------------------------------
DROP DATABASE IF EXISTS my_shop3_ex;
CREATE DATABASE my_shop3_ex;
USE my_shop3_ex;

-----------------------------------------
-- 테이블 초기화 (외래 키 종속성을 고려하여 자식 테이블부터 삭제)
-----------------------------------------
DROP TABLE IF EXISTS pay;
DROP TABLE IF EXISTS delivery;
DROP TABLE IF EXISTS order_item;
DROP TABLE IF EXISTS orders;
DROP TABLE IF EXISTS product;
DROP TABLE IF EXISTS member;

-- 1. member 테이블
CREATE TABLE member (
  member_id BIGINT  NOT NULL AUTO_INCREMENT, -- 회원 ID (PK)
  login_id VARCHAR(50) NOT NULL,    -- 로그인 ID
  password VARCHAR(255) NOT NULL,    -- 비밀번호 (암호화)
  member_name  VARCHAR(50) NOT NULL,    -- 이름
  email  VARCHAR(100) NOT NULL,    -- 이메일
  addr VARCHAR(255) NULL,     -- 주소
  created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  updated_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,

  PRIMARY KEY (member_id),
  UNIQUE KEY uq_login_id (login_id),
  UNIQUE KEY uq_email (email)
);

-- 2. product 테이블
CREATE TABLE product (
  product_id BIGINT  NOT NULL AUTO_INCREMENT, -- 상품 ID (PK)
  product_name  VARCHAR(100) NOT NULL,    -- 상품명
```

```sql
  product_price  INT   NOT NULL,    -- 가격
  stock_quantity INT   NOT NULL DEFAULT 0, -- 재고 수량
  created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  updated_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,

  PRIMARY KEY (product_id),
  INDEX idx_product_name (product_name) -- 상품명 검색용 인덱스
);

-- 3. orders 테이블
CREATE TABLE orders (
  order_id BIGINT  NOT NULL AUTO_INCREMENT, -- 주문 ID (PK)
  member_id BIGINT  NOT NULL,    -- 회원 ID (FK)
  ordered_at DATETIME NOT NULL,  -- 주문일 (애플리케이션에서 생성)
  order_status VARCHAR(20) NOT NULL DEFAULT 'ORDERED', -- 주문 상태
  total_amount INT   NOT NULL,    -- 총 주문 금액 (역정규화)
  created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  updated_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,

  PRIMARY KEY (order_id),
  CONSTRAINT fk_orders_member FOREIGN KEY (member_id)
    REFERENCES member (member_id),
  INDEX idx_order_status_ordered_at (order_status, ordered_at) -- 관리자용 주문 조
회 인덱스
);

-- 4. order_item 테이블
CREATE TABLE order_item (
  order_item_id BIGINT  NOT NULL AUTO_INCREMENT, -- 주문 상품 ID (PK)
  order_id BIGINT  NOT NULL,    -- 주문 ID (FK)
  product_id BIGINT  NOT NULL,    -- 상품 ID (FK)
  product_name VARCHAR(100) NOT NULL,    -- 주문 당시 상품명 (역정규화)
  order_price INT   NOT NULL,    -- 주문 당시 가격
  order_quantity  INT   NOT NULL,    -- 주문 수량
  created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  updated_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,

  PRIMARY KEY (order_item_id),
  CONSTRAINT fk_order_item_orders FOREIGN KEY (order_id)
    REFERENCES orders (order_id),
```

```sql
  CONSTRAINT fk_order_item_product FOREIGN KEY (product_id)
    REFERENCES product (product_id),

  -- [추가됨] 주문 ID와 상품 ID에 대한 유니크 제약 조건
  CONSTRAINT uq_order_id_product_id UNIQUE (order_id, product_id)
);

-- 5. delivery 테이블
CREATE TABLE delivery (
  delivery_id BIGINT  NOT NULL AUTO_INCREMENT, -- 배송 ID (PK)
  order_id  BIGINT  NOT NULL,    -- 주문 ID (FK, Unique)
  delivery_status  VARCHAR(20) NOT NULL DEFAULT 'READY',-- 배송 상태
  tracking_no VARCHAR(50) NULL,     -- 운송장 번호
  ship_addr VARCHAR(255) NOT NULL,    -- 배송지
  created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  updated_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,

  PRIMARY KEY (delivery_id),
  UNIQUE KEY uq_delivery_order_id (order_id), -- 주문 하나당 배송은 하나
  CONSTRAINT fk_delivery_orders FOREIGN KEY (order_id)
    REFERENCES orders (order_id)
);

-- 6. pay 테이블
CREATE TABLE pay (
  pay_id BIGINT  NOT NULL AUTO_INCREMENT, -- 결제 ID (PK)
  order_id BIGINT  NOT NULL,    -- 주문 ID (FK, Unique)
  pay_method VARCHAR(50) NOT NULL,    -- 결제 수단
  pay_amount  INT   NOT NULL,    -- 결제 금액
  pay_status  VARCHAR(20) NOT NULL,    -- 결제 상태
  paid_at DATETIME NULL,     -- 결제 완료일
  created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  updated_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,

  PRIMARY KEY (pay_id),
  UNIQUE KEY uq_pay_order_id (order_id), -- 주문 하나당 결제는 하나
  CONSTRAINT fk_pay_orders FOREIGN KEY (order_id)
    REFERENCES orders (order_id)
);
```

쇼핑몰 DB 구축 통합 스크립트 - 샘플 DML

```
-----------------------------------------
-- 샘플 데이터 입력 (DML)
-----------------------------------------

-- 1. 회원 데이터
-- 실제 서비스에서는 비밀번호를 반드시 암호화(해싱)하여 저장해야 한다.
INSERT INTO member(login_id, password, member_name, email, addr)
VALUES
('sejong', 'pass123!', '세종대왕', 'sejong@example.com', '서울시 종로구'),
('sunsin', 'pass456!', '이순신', 'sunsin@example.com', '전라남도 여수시'),
('curie', 'pass789!', '마리 퀴리', 'curie@example.com', '프랑스 파리'),
('nate', 'pass101!', '네이트', 'nate@example.com', '서울시 송파구'),
('jobs', 'pass112!', '스티브 잡스', 'jobs@example.com', '미국 캘리포니아'),
('sean', 'pass131!', '션', 'sean@example.com', '성남시 분당구');

-- 2. 상품 데이터 (IT 기기 및 전문 서적)
INSERT INTO product(product_name, product_price, stock_quantity)
VALUES
('싱크패드 노트북', 3500000, 15),
('리얼포스 키보드', 380000, 40),
('델 모니터', 1200000, 25),
('JPA 프로그래밍 책', 32000, 150),
('로지텍 지슈라2 마우스', 139000, 200),
('로지텍 웹캠', 450000, 30),
('갤럭시 S25 휴대폰', 1800000, 50),
('아이폰 17', 1900000, 50);


-- 3. 주문 데이터 & 관련 데이터 (주문, 주문상품, 배송, 결제)
-- 시나리오 1: 세종대왕, 'JPA 프로그래밍 책' 도서 1권 주문 (배송 완료)
INSERT INTO orders(member_id, ordered_at, order_status, total_amount)
VALUES
(1, '2025-09-05 10:00:00', 'COMPLETED', 32000);
SET @last_order_id = LAST_INSERT_ID();

INSERT INTO order_item(order_id, product_id, product_name, order_price,
order_quantity)
VALUES
(@last_order_id, 4, 'JPA 프로그래밍 책', 32000, 1);
```

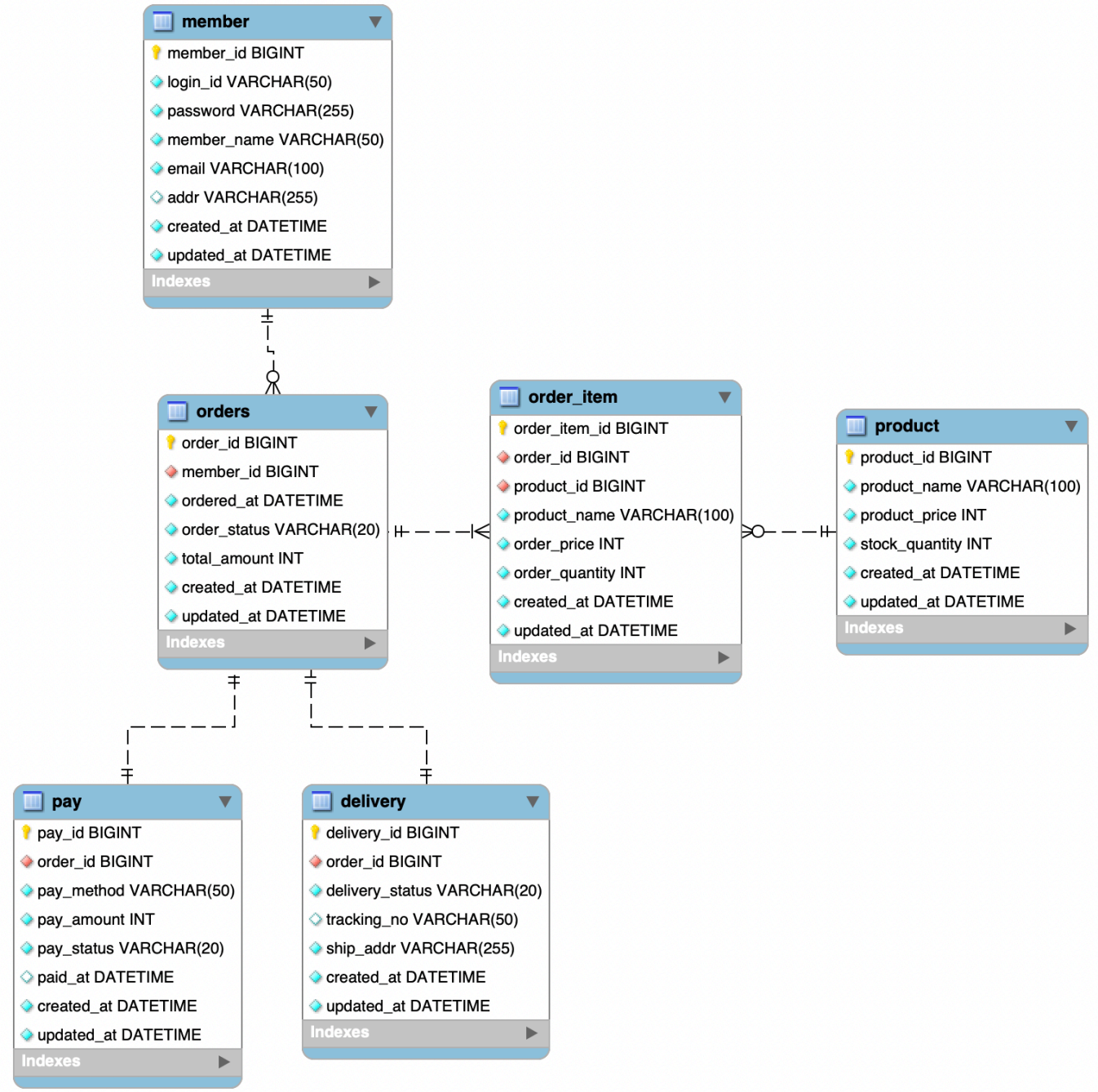
```sql
INSERT INTO delivery(order_id, delivery_status, tracking_no, ship_addr)
VALUES
(@last_order_id, 'COMPLETED', 'KR13970515', '서울시 종로구 경복궁');

INSERT INTO pay(order_id, pay_method, pay_amount, pay_status, paid_at)
VALUES
(@last_order_id, 'BANK_TRANSFER', 32000, 'PAID', '2025-09-05 10:05:00');


-- 시나리오 2: 이순신, 전투 지휘용 '델 모니터' 1대 주문 (배송중)
INSERT INTO orders(member_id, ordered_at, order_status, total_amount)
VALUES
(2, '2025-09-15 14:30:00', 'SHIPPING', 1200000);
SET @last_order_id = LAST_INSERT_ID();

INSERT INTO order_item(order_id, product_id, product_name, order_price,
order_quantity)
VALUES
(@last_order_id, 3, '델 모니터', 1200000, 1);

INSERT INTO delivery(order_id, delivery_status, tracking_no, ship_addr)
VALUES
(@last_order_id, 'SHIPPING', 'KR15450428', '전라남도 여수시 진남관');

INSERT INTO pay(order_id, pay_method, pay_amount, pay_status, paid_at)
VALUES
(@last_order_id, 'CARD', 1200000, 'PAID', '2025-09-15 14:32:00');


-- 시나리오 3: 네이트, '싱크패드 노트북'과 '리얼포스 키보드' 주문 (배송 준비)
INSERT INTO orders(member_id, ordered_at, order_status, total_amount)
VALUES
(4, '2025-09-20 09:00:00', 'ORDERED', 3880000);
SET @last_order_id = LAST_INSERT_ID();

INSERT INTO order_item(order_id, product_id, product_name, order_price,
order_quantity)
VALUES
(@last_order_id, 1, '싱크패드 노트북', 3500000, 1),
(@last_order_id, 2, '리얼포스 키보드', 380000, 1);

INSERT INTO delivery(order_id, delivery_status, ship_addr)
```

```sql
VALUES
(@last_order_id, 'READY', '서울시 송파구 잠실동');

INSERT INTO pay(order_id, pay_method, pay_amount, pay_status, paid_at)
VALUES
(@last_order_id, 'CARD', 3880000, 'PAID', '2025-09-20 09:01:00');


-- 시나리오 4: 스티브 잡스, '갤럭시 S25 휴대폰'을 주문했으나 취소
INSERT INTO orders(member_id, ordered_at, order_status, total_amount)
VALUES
(5, '2025-09-11 18:00:00', 'CANCELED', 1800000);
SET @last_order_id = LAST_INSERT_ID();

INSERT INTO order_item(order_id, product_id, product_name, order_price,
order_quantity)
VALUES
(@last_order_id, 7, '갤럭시 S25 휴대폰', 1800000, 1);

INSERT INTO delivery(order_id, delivery_status, tracking_no, ship_addr)
VALUES
(@last_order_id, 'SHIPPING', 'EA12341234', '미국 캘리포니아');

INSERT INTO pay(order_id, pay_method, pay_amount, pay_status)
VALUES
(@last_order_id, 'CARD', 1800000, 'CANCELED');


-- 시나리오 5: 션, 'JPA 프로그래밍 책'과 '로지텍 지슈라2 마우스' 주문 (배송 완료)
INSERT INTO orders(member_id, ordered_at, order_status, total_amount)
VALUES
(6, '2025-08-25 21:00:00', 'COMPLETED', 171000);
SET @last_order_id = LAST_INSERT_ID();

INSERT INTO order_item(order_id, product_id, product_name, order_price,
order_quantity)
VALUES
(@last_order_id, 4, 'JPA 프로그래밍 책', 32000, 1),
(@last_order_id, 5, '로지텍 지슈라2 마우스', 139000, 1);

INSERT INTO delivery(order_id, delivery_status, tracking_no, ship_addr)
VALUES
(@last_order_id, 'COMPLETED', 'KR19691228', '성남시 분당구 판교동');
```

```
INSERT INTO pay(order_id, pay_method, pay_amount, pay_status, paid_at)
VALUES
(@last_order_id, 'CARD', 171000, 'PAID', '2025-08-25 21:05:00');


-- 시나리오 6: 션, '싱크패드 노트북'과 '로지텍 웹캠' 추가 주문 (배송중)
INSERT INTO orders(member_id, ordered_at, order_status, total_amount)
VALUES
(6, '2025-09-22 13:10:00', 'SHIPPING', 3950000);
SET @last_order_id = LAST_INSERT_ID();

INSERT INTO order_item(order_id, product_id, product_name, order_price,
order_quantity)
VALUES
(@last_order_id, 1, '싱크패드 노트북', 3500000, 1),
(@last_order_id, 6, '로지텍 웹캠', 450000, 1);

INSERT INTO delivery(order_id, delivery_status, tracking_no, ship_addr)
VALUES
(@last_order_id, 'SHIPPING', 'KR20250922', '성남시 분당구 정자동');

INSERT INTO pay(order_id, pay_method, pay_amount, pay_status, paid_at)
VALUES
(@last_order_id, 'CARD', 3950000, 'PAID', '2025-09-22 13:11:00');
```

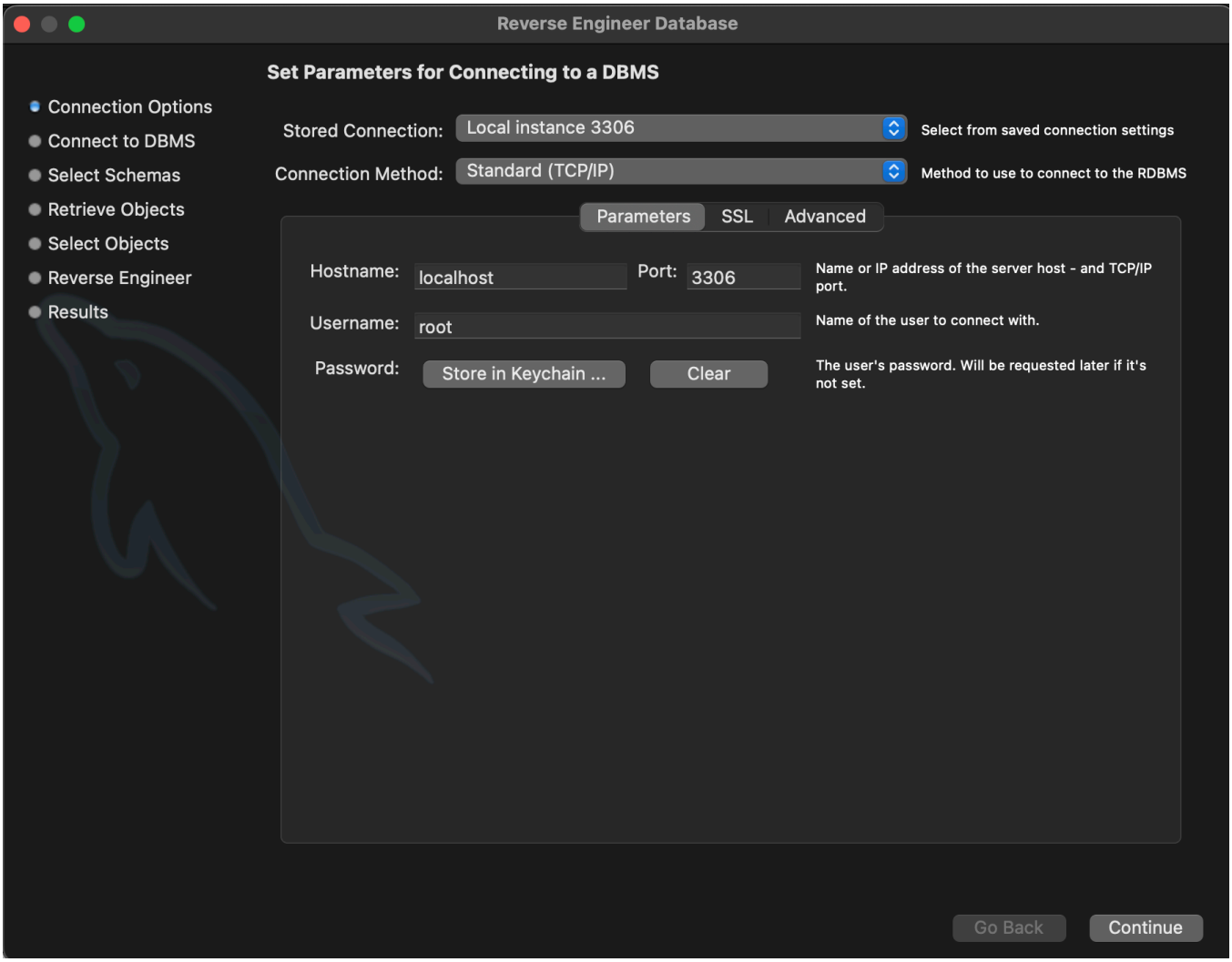
물리적 모델 - ERD 자동 생성

MySQL Workbench를 사용하면 우리가 만든 테이블을 기반으로 물리적 모델의 ERD를 그림과 같이 생성할 수 있다.

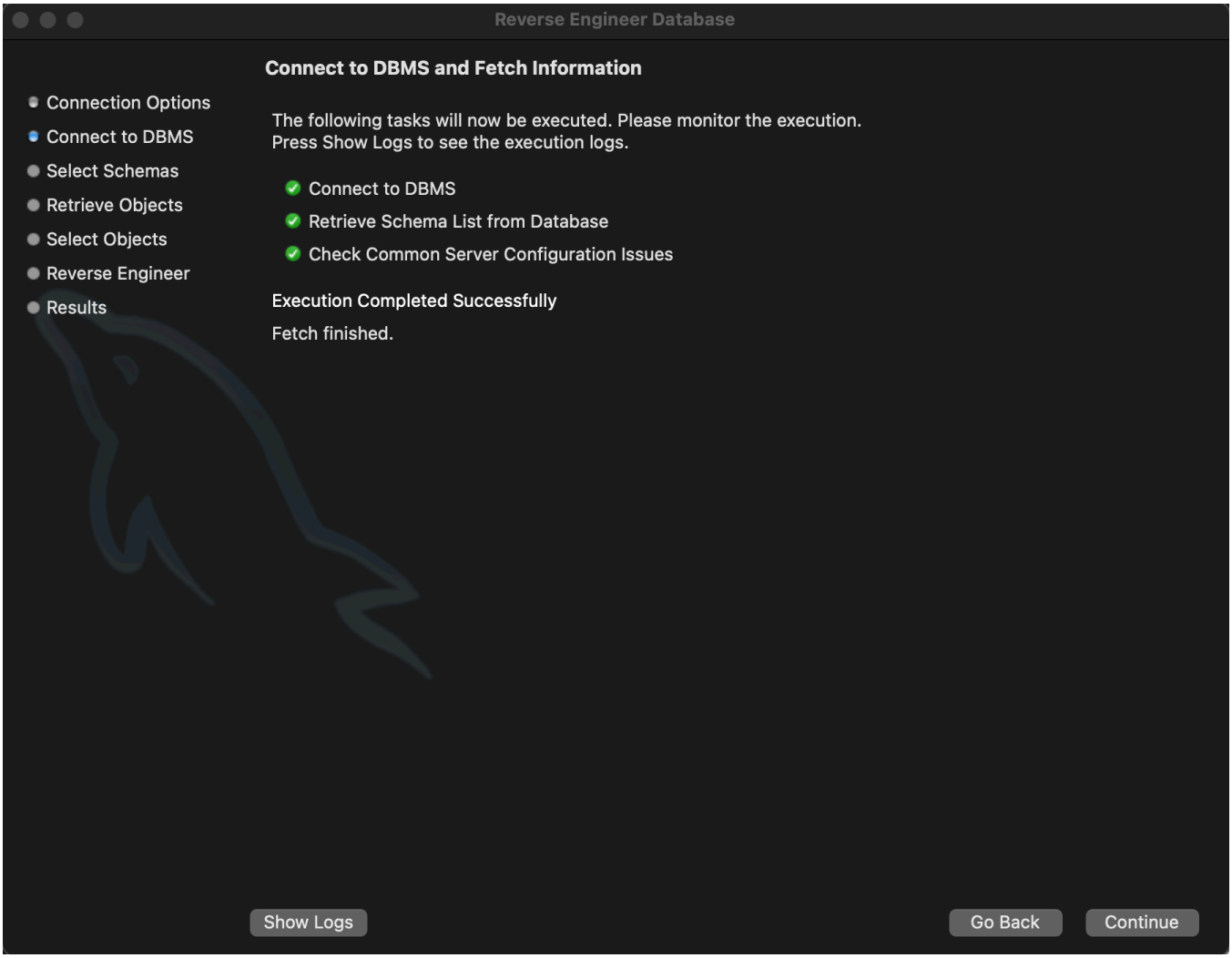


MySQL 워크벤치 ERD 생성

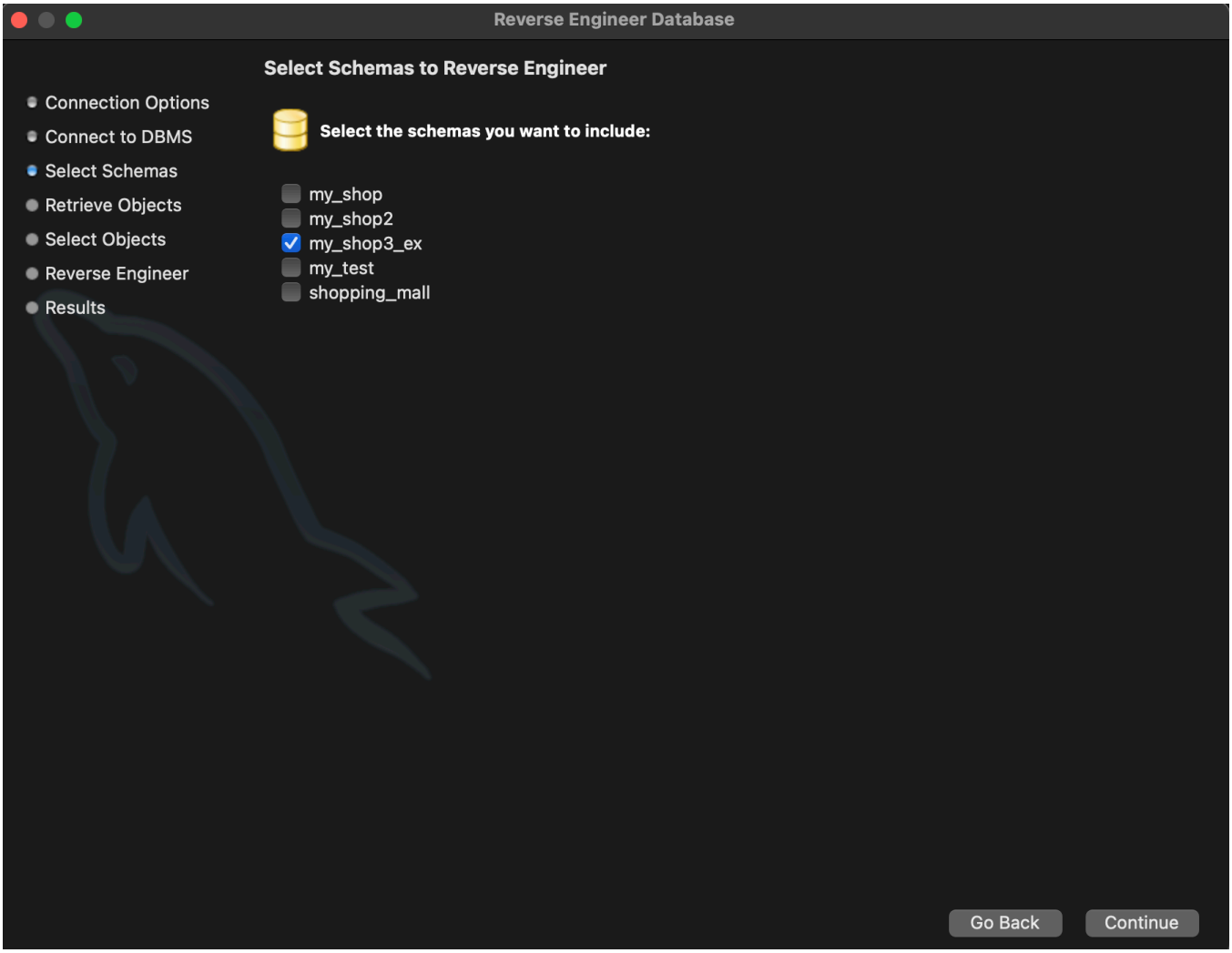
- 메뉴 ➜ Database ➜ Reverse Engineer...을 선택하자.



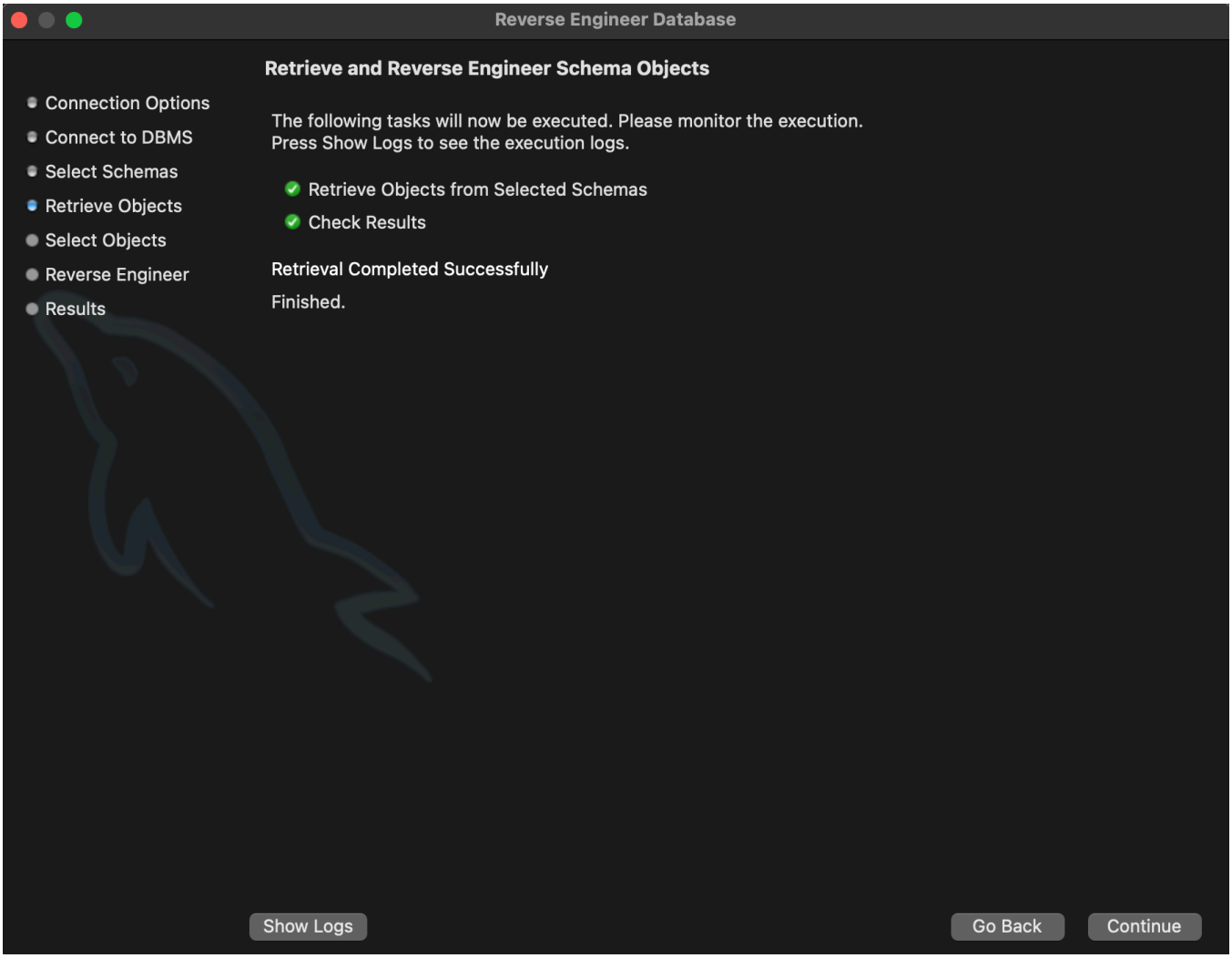
- Continue를 선택한다.



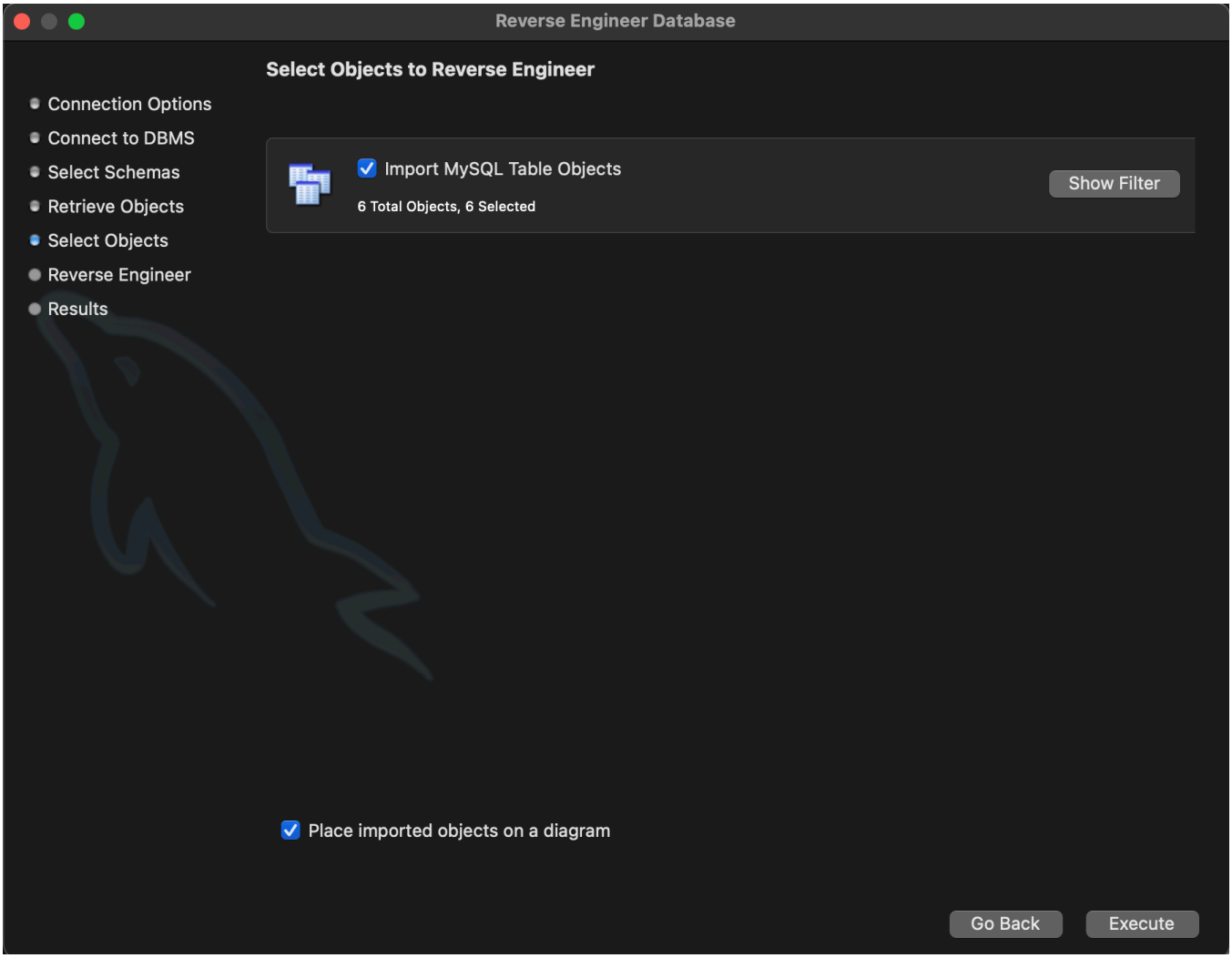
- Continue를 선택한다.



- ERD를 만들 데이터베이스(`my_shop3_ex`)를 체크한 다음 Continue를 선택한다.



- Continue를 선택한다.



- Execute를 선택한다.
- 참고: 원하는 테이블을 직접 선택하고 싶다면 Show Filter에서 수정하면 된다.

이후에 결과를 알려주는데, Continue를 선택하면 테이블을 기반으로 ERD가 완성된다.

> **참고**
> 테이블을 기반으로 ERD를 그려주는 기능은 완벽하지는 않다.
> 일대다, 일대일, 필수적, 선택적 관계 등은 관계의 라인을 더블 클릭해서 직접 수정해야 한다.
> 만약 FK를 설정하지 않았다면 관계가 표현되지 않는다.

쇼핑몰 기능 확인1

우리가 설계한 테이블과 인덱스가 실제 서비스 기능에서 어떻게 동작하고, 정말로 성능에 도움이 되는지 직접 확인해 보

자. MySQL의 `EXPLAIN` 명령어를 사용하면, 데이터베이스가 우리의 SQL 쿼리를 어떤 방식으로 실행할 계획인지 미리 엿볼 수 있다. 이를 통해 우리가 만든 인덱스가 의도대로 잘 사용되는지 검증할 수 있다.

각각의 시나리오 별로 잘 작동하는지 확인해보자.

회원 로그인 처리

상황: 사용자가 ID로 'sejong'을 입력하고 로그인을 시도한다. 시스템은 이 ID에 해당하는 회원 정보를 데이터베이스에서 조회하여 비밀번호를 검증해야 한다.

```
EXPLAIN
SELECT
    member_id,
    login_id,
    password, -- 애플리케이션에서 비밀번호 검증을 위해 사용될 암호화된 비밀번호
    member_name
FROM
    member
WHERE
    login_id = 'sejong';
```

[실행 계획]

id	table	type	key	ref	rows	filtered	Extra
1	member	const	uq_login_id	const	1	100.00	

분석

1. `table`: `member` 테이블을 사용했다.
2. `type`: `const`. `login_id` 컬럼은 `UNIQUE` 키이므로, MySQL은 이 쿼리를 최적화하는 단계에서 값을 상수('sejong')로 취급하고 단 하나의 행만 읽으면 된다는 것을 안다. 성능이 매우 우수함을 의미한다.
3. `key`: `uq_login_id`. `login_id` 컬럼에 설정한 `UNIQUE` 제약조건으로 인해 자동으로 생성된 인덱스를 정확하게 사용했다.
4. `rows`: `1`. MySQL이 단 1개의 행만 찾으면 쿼리가 종료될 것이라고 완벽하게 예측했다.
5. **결론**: 이 실행 계획은 로그인이 얼마나 효율적으로 처리되는지를 명확히 보여준다. 전체 회원 수가 100만 명, 1000만 명이 되더라도 로그인 속도는 거의 동일하게 유지될 것이다.

해설 및 실무 Tip

- 로그인은 웹사이트에서 가장 빈번하게 발생하는 쿼리 중 하나이다. `login_id` 컬럼에 `UNIQUE` 제약조건을 설정했기 때문에 자동으로 고유 인덱스가 생성되었고, 이 덕분에 매우 빠르게 특정 사용자를 찾아낼 수 있다.
- **매우 중요한 보안 사항**: 데이터베이스에 저장된 `password` 값은 'pass123!'과 같은 평문이 아닌, 암호화된 해시(Hash) 값이어야 한다. 실제 애플리케이션에서는 사용자에게 입력받은 비밀번호를 동일한 해시 함수로 암호화한 뒤, 데이터베이스에 저장된 해시 값과 일치하는지 비교하는 방식으로 인증을 처리한다. 절대로 평문 비밀번호를 데이터베이스에 저장해서는 안 된다.

```
SELECT
    member_id,
    login_id,
    password,
    member_name
FROM
    member
WHERE
    login_id = 'sejong';
```

[실행 결과]

member_id	login_id	password	member_name
1	sejong	pass123!	세종대왕

회원 주문 목록 조회

상황: '션'(member_id = 6) 회원이 마이페이지에서 자신의 주문 내역을 최신순으로 조회한다.

```
EXPLAIN
SELECT order_id, ordered_at, order_status, total_amount
FROM orders
WHERE member_id = 6
ORDER BY ordered_at DESC;
```

[실행 계획]

id	table	type	key	ref	rows	filtered	Extra
1	orders	ref	fk_orders_member	const	2	100.00	Using filesort

분석

1. `table`: `orders` 테이블을 사용했다.
2. `type`: `ref`. 인덱스를 사용하여 동등 비교(`=`) 조건으로 데이터를 찾는 효율적인 방식이다.
3. `key`: `fk_orders_member`. 우리가 예상한 대로 `member_id` 컬럼의 외래 키에 자동으로 생성된 인덱스를 잘 사용하고 있다.
4. `rows`: `2`. MySQL이 약 2개의 행만 읽으면 될 것이라고 예측했다. (실제 데이터 건수에 따라 달라질 수 있음) 전체 주문 데이터가 아무리 많아져도, 이 인덱스 덕분에 특정 회원의 주문만 매우 빠르게 찾아낼 수 있다.
5. `Extra`: `Using filesort`. `WHERE` 절로 데이터를 찾은 후, `ORDER BY ordered_at DESC` 정렬을 위해 별도의 정렬 작업(filesort)을 수행했다는 의미다. 데이터 양이 적어서 큰 문제는 아니지만, 만약 이 기능이 매우 중요하고 회원의 주문 건수가 아주 많다면 `(member_id, ordered_at)` 복합 인덱스를 고려해볼 수도 있다. 하지만 지금 해당 기능은 과잉 최적화이므로 현재 상태를 유지하자.

```sql
SELECT order_id, ordered_at, order_status, total_amount
FROM orders
WHERE member_id = 6
ORDER BY ordered_at DESC;
```

[실행 결과]

order_id	ordered_at	order_status	total_amount
6	2025-09-22 13:10:00	SHIPPING	3950000
5	2025-08-25 21:00:00	COMPLETED	171000

회원 주문 목록 조회 (역정규화 효과)

앞서 본 '회원 주문 목록 조회' 쿼리가 간단해 보이지만, 여기에는 `total_amount` **역정규화**라는 중요한 설계 결정이 숨어있다. `orders` 테이블에 `total_amount`가 없었다면, 주문 목록을 조회할 때마다 매번 총액을 계산해야 했을 것이다.

만약 역정규화를 하지 않았다면 다음과 같은 복잡하고 비효율적인 쿼리를 실행해야 한다.

```
-- 역정규화를 하지 않았다면 매번 실행해야 했을 복잡한 쿼리
SELECT
    o.order_id,
    o.ordered_at,
    o.order_status,
    SUM(oi.order_price * oi.order_quantity) AS calculated_total_amount
FROM
    orders o
JOIN
    order_item oi ON o.order_id = oi.order_id
WHERE
    o.member_id = 6
GROUP BY
    o.order_id, o.ordered_at, o.order_status
ORDER BY
    o.ordered_at DESC;
```

위 쿼리는 주문 목록에 표시될 모든 주문 건에 대해 `order_item` 테이블을 `JOIN`하고 `GROUP BY`를 통해 합계를 계산한다. 주문 내역 조회는 사용자가 매우 빈번하게 사용하는 기능이므로, 매번 이렇게 복잡한 연산을 수행하는 것은 데이터베이스에 큰 부하를 준다.

`orders` 테이블에 `total_amount`를 역정규화함으로써, 우리는 `JOIN`**과 집계 연산을 피하고** 단순히 `orders` 테이블만 읽어서 매우 빠르고 효율적으로 주문 목록을 표시할 수 있다. 이것이 바로 읽기 성능을 최적화하기 위한 역정규화의 강력한 효과다.

[실행 결과]

order_id	ordered_at	order_status	total_amount
6	2025-09-22 13:10:00	SHIPPING	3950000
5	2025-08-25 21:00:00	COMPLETED	171000

상품명 검색

상황: 사용자가 검색창에 'JPA'라는 키워드를 입력하여 상품을 찾는다.

```
EXPLAIN
SELECT product_id, product_name, product_price
FROM product
WHERE product_name LIKE 'JPA%';
```

[실행 계획]

id	table	type	key	rows	filtered	Extra
1	product	range	idx_product_name	1	100.00	Using index condition

분석

1. `type`: `range`. 인덱스를 사용하여 특정 범위의 데이터를 검색했다는 의미다. `LIKE 'JPA%'` 는 'JPA'로 시작하는 모든 문자열 범위를 검색하므로 `range` 타입으로 동작한다.
2. `key`: `idx_product_name`. 우리가 상품명 검색을 위해 `product_name` 컬럼에 생성한 `idx_product_name` 인덱스가 정확하게 사용되었다.
3. `rows`: `1`. MySQL이 약 1개의 행만 스캔하면 된다고 예측했다. 인덱스가 없었다면 모든 상품 데이터를 스캔해야 했을 것이다.
4. `Extra`: `Using index condition`. 인덱스를 통해 데이터를 필터링하여 불필요한 데이터 접근을 줄였다는 긍정적인 신호다.

```
SELECT product_id, product_name, product_price
FROM product
WHERE product_name LIKE 'JPA%';
```

[실행 결과]

product_id	product_name	product_price
4	JPA 프로그래밍 책	32000

관리자 주문 조회 (복합 인덱스)

상황: 관리자가 어드민 페이지에서 '2025년 9월에 취소(`CANCELED`)된 주문'을 조회한다.

```
EXPLAIN
SELECT order_id, member_id, ordered_at, order_status
FROM orders
WHERE order_status = 'CANCELED'
  AND ordered_at BETWEEN '2025-09-01 00:00:00' AND '2025-09-30 23:59:59';
```

[실행 계획]

id	table	type	key	rows	filtered	Extra
1	orders	range	idx_order_status_ordered_at	1	100	Using index condition

분석

1. `type`: `range`. `order_status`는 등호(=) 조건이지만, `ordered_at`이 범위(BETWEEN) 조건이므로 최종적으로 `range` 타입으로 동작했다.
2. `key`: `idx_order_status_ordered_at`. 우리가 관리자 조회 성능을 위해 `(order_status, ordered_at)` 순서로 만든 복합 인덱스가 정확히 사용되었다.
3. `rows`: `1`. MySQL은 인덱스를 사용해 `order_status`가 'CANCELED'인 것들 중에서 `ordered_at`이 조건에 맞는 데이터만 효율적으로 찾아 약 1건만 읽으면 된다고 예측했다. 이 복합 인덱스가 없었다면 훨씬 많은 데이터를 스캔해야 했을 것이다.

```
SELECT order_id, member_id, ordered_at, order_status
FROM orders
WHERE order_status = 'CANCELED'
  AND ordered_at BETWEEN '2025-09-01 00:00:00' AND '2025-09-30 23:59:59';
```

[실행 결과]

order_id	member_id	ordered_at	order_status
4	5	2025-09-11 18:00:00	CANCELED

주문 상세 조회 (역정규화 효과)

상황: 한 회원이 자신의 주문 상세 내역을 본다. '네이트' 회원의 주문(`order_id` = 3)에 어떤 상품들이 있었는지 확인한다.

```sql
-- product 테이블과 JOIN 할 필요가 없다.
SELECT order_item_id, product_name, order_price, order_quantity
FROM order_item
WHERE order_id = 3;
```

[실행 결과]

order_item_id	product_name	order_price	order_quantity
3	싱크패드 노트북	3500000	1
4	리얼포스 키보드	380000	1

분석

이 쿼리는 `order_id`가 `order_item` 테이블의 외래 키이므로 자동으로 생성된 인덱스를 사용해 매우 빠르게 동작한다. 여기서 핵심은 `EXPLAIN` 결과보다도 SQL 쿼리 자체에 있다. 만약 우리가 `product_name`을 역정규화하지 않았다면, 상품명을 가져오기 위해 다음과 같이 항상 `product` 테이블과 `JOIN`을 해야만 했을 것이다.

```sql
-- 역정규화를 하지 않았다면 매번 실행해야 했을 쿼리
SELECT
    oi.order_item_id,
    p.product_name,
```

```
    oi.order_price,
    oi.order_quantity
FROM
    order_item oi
JOIN
    product p ON oi.product_id = p.product_id
WHERE
    oi.order_id = 3;
```

주문 상세 조회는 매우 빈번한 기능이다. 역정규화를 통해 이 `JOIN` 연산을 제거함으로써 우리는 데이터베이스의 부하를 크게 줄이고 서비스의 응답 속도를 향상시키는, 최적화를 수행한 것이다.

쇼핑몰 기능 확인2

비즈니스 분석과 마케팅에 필요한 통계 기능, 시스템 운영을 위한 관리 기능을 확인해보자.

월별 매출 통계 리포트

상황: 경영진 보고를 위해 월별 총 주문 건수와 매출액을 집계해야 한다.

```
SELECT
    DATE_FORMAT(ordered_at, '%Y-%m') AS `year_month`,
    COUNT(order_id) AS total_orders,
    SUM(total_amount) AS total_sales
FROM
    orders
WHERE
    order_status NOT IN ('CANCELED') -- 취소된 주문은 매출에서 제외
GROUP BY
    `year_month`
ORDER BY
    `year_month`;
```

> **MySQL에서 year_month는 예약어이다.**

> 백틱(`)으로 감싸서 사용하자.

해설

`DATE_FORMAT` 함수를 사용하면 `DATETIME` 형식의 데이터를 '년-월' 또는 '일자' 등 원하는 형식으로 바꿔서 그룹핑 할 수 있다.
참고로 WHERE 절에서 취소된 주문을 제외하는 것은 정확한 매출 통계를 위한 필수 조건이다.

[실행 결과]

year_month	total_orders	total_sales
2025-08	1	171000
2025-09	4	9062000

마케팅 - 베스트셀러 상품 TOP 5

상황: 2025년 8월 ~ 9월에 가장 많이 팔린 상품 5개를 판매량 순으로 확인하여 마케팅에 활용하고 싶다.

```sql
-- 판매량 기준 TOP 5
SELECT
    oi.product_name,
    SUM(oi.order_quantity) AS total_quantity_sold,
    SUM(oi.order_price * oi.order_quantity) AS total_sales_amount
FROM
    order_item oi
JOIN
    orders o ON oi.order_id = o.order_id
WHERE
    o.order_status NOT IN ('CANCELED')
AND
    o.ordered_at BETWEEN '2025-08-01 00:00:00' AND '2025-09-30 23:59:59'
GROUP BY
    oi.product_name
ORDER BY
    total_quantity_sold DESC, total_sales_amount DESC
LIMIT 5;
```

해설

여러 테이블을 조인하여 복합적인 분석을 수행하는 쿼리이다.

`order_item`의 판매량(`order_quantity`)과 판매가(`order_price`)를 SUM 함수로 집계한다. `product` 테이블과 조인할 필요 없이 역정규화된 `product_name`을 바로 사용해서 더 효율적이다.

[실행 결과]

판매량이 동일한 경우 매출액 순으로 정렬된다.

product_name	total_quantity_sold	total_sales_amount
싱크패드 노트북	2	7000000
JPA 프로그래밍 책	2	64000
델 모니터	1	1200000
로지텍 웹캠	1	450000
리얼포스 키보드	1	380000

고객 관리 - VIP 고객 식별

상황: 총 구매액이 가장 높은 상위 2명의 고객에게 감사의 의미로 쿠폰을 발송하려고 한다.

```
SELECT
    m.member_id,
    m.member_name,
    m.email,
    SUM(o.total_amount) AS total_purchase_amount,
    COUNT(o.order_id) AS total_order_count
FROM
    member m
JOIN
    orders o ON m.member_id = o.member_id
WHERE
    o.order_status NOT IN ('CANCELED')
GROUP BY
    m.member_id, m.member_name, m.email
ORDER BY
```

```
    total_purchase_amount DESC
LIMIT 2;
```

해설

고객(`member`)과 주문(`orders`) 테이블을 조인하여 고객별로 총 구매액을 집계한다. CRM(고객 관계 관리)에서 가장 기본적이면서도 중요한 분석 기법이다.

[실행 결과]

member_id	member_name	email	total_purchase_amount	total_order_count
6	션	sean@example.com	4121000	2
4	네이트	nate@example.com	3880000	1

재고 부족 상품 확인

상황: 재고가 50개 미만으로 남은 상품 목록을 확인하여 재주문 프로세스를 시작해야 한다.

```
SELECT
    product_id,
    product_name,
    stock_quantity
FROM
    product
WHERE
    stock_quantity < 50
ORDER BY
    stock_quantity ASC;
```

해설

재고 관리 시스템의 핵심 기능이다. 스케줄러를 이용해 이 쿼리를 주기적으로 실행하고, 결과가 있을 경우 담당자에게 알림을 보내는 방식으로 자동화할 수 있다.

[실행 결과]

product_id	product_name	stock_quantity
1	싱크패드 노트북	15
3	델 모니터	25
6	로지텍 웹캠	30
2	리얼포스 키보드	40

정리

물리적 모델링 - 실습 시작

- 데이터베이스 설계의 마지막 단계로, 논리적 모델을 실제 데이터베이스(MySQL)에 맞는 물리적 스키마로 변환한다.
- 물리적 모델링의 최종 산출물은 **테이블 정의서**와 실제 테이블을 생성하는 **DDL(CREATE TABLE)** 스크립트이다.
- 핵심 목표는 **성능 최적화**이며, 이를 위해 **인덱스 설계**와 **역정규화** 기법을 적용한다.
- 물리적 모델링은 테이블 변환, 데이터 타입 정의, 제약 조건 설정, 인덱스 설계, 역정규화 순서로 진행한다.

인덱스 설계 - 실습

- 인덱스는 데이터 조회 속도를 높이는 '목차'와 같으며, 없으면 모든 데이터를 스캔하는 **풀 테이블 스캔(Full Table Scan)**이 발생한다.
- MySQL은 PK, UNIQUE, FK 제약 조건에 대해 자동으로 인덱스를 생성한다.
- WHERE 절에 자주 사용되거나 JOIN의 연결고리가 되는 컬럼에 인덱스를 생성한다.
- 두 개 이상의 컬럼이 WHERE 절에 함께 자주 사용될 경우 **복합 인덱스**를 고려하며, **컬럼 순서가 매우 중요**하다. (등호 조건 ➜ 범위 조건 순)
- 선택도가 높은 컬럼의 인덱스만으로 충분히 데이터가 걸러진다면, 추가적인 복합 인덱스는 오히려 쓰기(INSERT, UPDATE, DELETE) 성능을 저하시키는 **과잉 최적화**가 될 수 있으므로 신중해야 한다.

역정규화 - 실습

- 역정규화는 정규화 원칙을 위배하여 조회 성능을 높이는 기법으로, 최후의 수단으로 고려해야 한다.

- **중복 컬럼 추가**: 잦은 JOIN을 피하기 위해 `order_item` 테이블에 `product_name`을 추가한다. 이는 주문 당시의 정보를 보존하는 **'스냅샷'** 데이터로서 비즈니스 요구사항도 만족시킨다.
- **파생 컬럼 추가**: 매번 계산해야 하는 값을 미리 계산해 저장한다. `orders` 테이블에 `total_amount`(총 주문 금액)를 추가하여 복잡한 집계 연산을 피한다.
- **테이블 통합**: `orders`와 `delivery`처럼 1:1 관계라도 **데이터의 생명주기와 변경 빈도가 다르면 통합하지 않는 것**이 좋다. 분리된 구조가 유연성과 확장성 측면에서 더 유리하다.

쇼핑몰 테이블 정의서

- 논리적 모델을 기반으로 컬럼의 영문명, 데이터 타입, 제약 조건 등을 구체적으로 명시한 문서이다.
- 모든 테이블에 데이터의 생성 및 수정 이력을 추적하기 위한 감사 컬럼 `created_at`**(생성일)**과 `updated_at`**(수정일)**을 추가하는 것이 표준이다.
- **시스템의 시간(**`created_at`**)**과 **비즈니스의 시간(**`ordered_at`**)**을 명확히 구분해야 한다. 시스템 시간은 데이터가 DB에 물리적으로 저장된 시각이며, 비즈니스 시간은 실제 주문이 발생한 시각을 의미한다.

쇼핑몰 DDL과 DB 만들기

- 테이블 정의서를 바탕으로 실제 데이터베이스에 테이블을 생성하는 SQL(`CREATE TABLE`) 스크립트이다.
- 인덱스 설계와 역정규화 전략이 모두 반영된 최종 코드로 구성된다.
- 데이터베이스 초기화(DROP/CREATE), 테이블 생성(DDL), 샘플 데이터 입력(DML)을 포함하는 통합 스크립트이다.
- 이 스크립트 하나로 언제든지 동일한 구조와 데이터를 가진 개발 및 테스트 환경을 구축할 수 있다.

물리적 모델 - ERD 자동 생성

- MySQL Workbench와 같은 툴을 사용하면 생성된 테이블 구조를 바탕으로 물리적 모델의 ERD를 자동으로 생성하여 시각적으로 확인할 수 있다.

쇼핑몰 기능 확인1

- `EXPLAIN` 명령어를 통해 SQL 쿼리의 실행 계획을 분석하여, 설계한 인덱스가 의도대로 효율적으로 사용되는지 검증한다.
- 로그인, 회원 주문 목록 조회, 상품 검색, 관리자 주문 조회 등 핵심 기능에서 인덱스가 잘 동작함을 확인한다.
- 역정규화를 통해 `JOIN`이나 복잡한 계산 없이 단순한 쿼리로 빠른 조회가 가능해진 효과를 확인한다.

쇼핑몰 기능 확인2

- 월별 매출 통계, 베스트셀러 상품, VIP 고객 식별, 재고 부족 상품 확인 등 비즈니스 분석 및 운영에 필요한 복잡한 조회 쿼리 예시를 다루었다.
- 잘 설계된 테이블 구조는 복잡한 통계 및 분석 쿼리도 효율적으로 처리할 수 있는 기반이 된다.