2. 조인1 - 내부 조인

#0.강의/2.데이터베이스로드맵/2.기본

실습 데이터 준비

데이터베이스를 본격적으로 사용하기 전에 아직 MySQL이 설치되어 있지 않다면 다음을 참고해서 본인의 환경에 맞는 MySQL을 설치하고, MySQL 워크벤치도 실행하자.

1.강의 소개와 수업 자료

- 윈도우 - MySQL 설치 안내
- macOS - MySQL 설치 안내

> **데이터베이스 입문 강의를 진행했다면 MySQL이 이미 설치되어 있다.**

앞서 데이터베이스 입문 강의를 통해 기본적인 SQL을 다룰 수 있게 되었을 것이다.
이번 강의에서는 실무에서 데이터를 다룰 때 반드시 알아야 할 핵심적인 기술들을 깊이 있게 파고든다.

우리의 성공하는 쇼핑몰 스타트업을 위한 테이블과 데이터를 지금부터 함께 만들어 보자.
실습을 시작하기 전에, 앞으로 사용할 데이터를 먼저 준비하자.

주요 비즈니스 규칙 및 제약사항

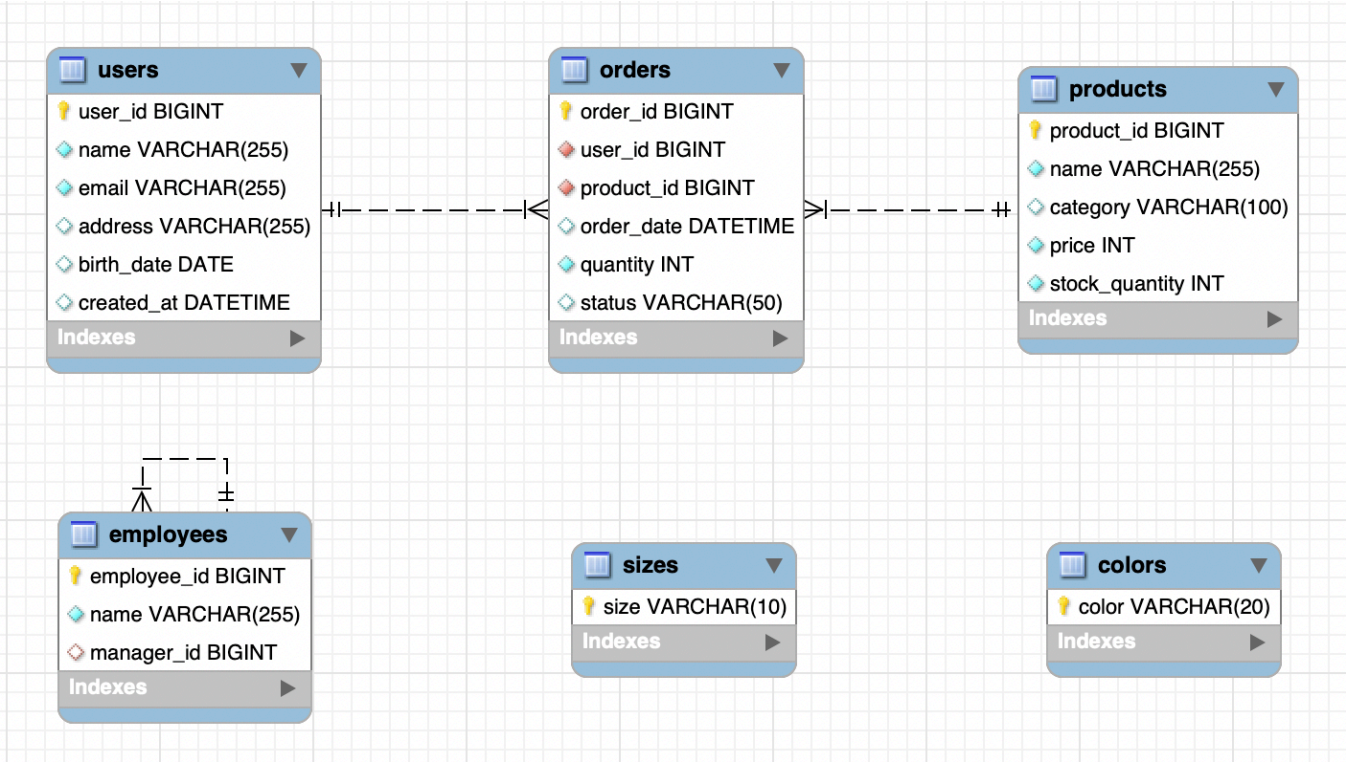
우리가 만들 쇼핑몰의 비즈니스 규칙은 다음과 같다.

1. **고객 가입**: 모든 고객은 고유한 이메일 주소를 가져야 한다. 이름과 이메일은 필수 정보다.
2. **주문 생성**: 주문은 반드시 특정 고객(`user_id`)과 특정 상품(`product_id`)에 연결되어야 한다.

    - 하나의 주문에 한 종류의 상품만 선택할 수 있다. 상품의 수량은 선택할 수 있다.
3. **주문 상태 관리**: 주문이 생성되면 기본 상태는 'PENDING'이며, 이후 'COMPLETED', 'SHIPPED', 'CANCELLED'로 변경될 수 있다.
    - PENDING(대기)
    - SHIPPED(배송)
    - COMPLETED(완료)
    - CANCELLED(취소)
4. **재고 관리**: 주문이 발생하면 해당 `products` 테이블의 `stock_quantity`(재고)는 주문 `quantity`(수량)만큼 차감되어야 한다.
    - 이 로직은 데이터베이스가 아니라 애플리케이션에서 구현해야 한다.
5. **직원 관리 구조**: 직원은 매니저를 가질 수 있으며, 매니저 또한 직원이다. 매니저가 없는 최상위 직원이 존재할 수 있다.

> **왜 이런 비즈니스 규칙과 제약사항을 먼저 살펴볼까?**
> 실제로 데이터베이스를 만들 때는 먼저 어떤 데이터가 필요하고, 그 데이터들이 어떻게 연결되는지 설계하는 과정이 꼭 필요하다.
> 하지만 이번 강의는 데이터베이스 설계가 주제가 아니기 때문에, 이런 규칙과 구조가 있다는 것만 간단히 이해하고 넘어가자.
> 데이터베이스 설계에 관한 부분은 데이터베이스 설계 강의에서 자세히 다룬다.

데이터 모델 다이어그램 (ERD)



고객 주문 상품의 관계

- **고객(users)**은 여러 개의 **주문(orders)**을 생성할 수 있다.
- **상품(products)**은 여러 **주문(orders)**에 포함될 수 있다.
- 하나의 **주문(orders)**은 한 명의 **고객**과 하나의 **상품**에 연결된다.

나머지 관계

- **직원(employees)**은 다른 **직원**을 관리하는 계층 구조(SELF JOIN)를 가진다.
  - `manager_id`를 통해 상사를 알 수 있다.
- **사이즈(sizes)**와 **색상(colors)** 테이블은 상품의 모든 옵션 조합을 생성하기 위해 사용된다.

테이블별 상세 요구사항

고객 (users) 테이블

고객의 개인 정보 및 계정 정보를 저장한다.

컬럼명	데이터 타입	제약 조건	설명
`user_id`	`BIGINT`	`PK`, `AUTO_INCREMENT`	고객의 고유 식별자 (자동 증가)
`name`	`VARCHAR(255)`	`NOT NULL`	고객의 이름 (필수 입력)
`email`	`VARCHAR(255)`	`NOT NULL`, `UNIQUE`	고객의 이메일. 로그인 ID로 사용 가능하며, 중복될 수 없음 (필수 입력)
`address`	`VARCHAR(255)`		고객의 주소 (선택 입력)
`birth_date`	`DATE`		고객의 생년월일 (선택 입력, YYYY-MM-DD 형식)
`created_at`	`DATETIME`	`DEFAULT CURRENT_TIMESTAMP`	고객 정보 생성 일시 (자동으로 현재 시간 기록)

상품 (products) 테이블

판매하는 상품의 정보를 관리한다.

컬럼명	데이터 타입	제약 조건	설명

product_id	BIGINT	PK, AUTO_INCREMENT	상품의 고유 식별자 (자동 증가)
name	VARCHAR(255)	NOT NULL	상품의 이름 (필수 입력)
category	VARCHAR(100)		상품의 카테고리 (선택 입력)
price	INT	NOT NULL	상품의 가격 (필수 입력)
stock_quantity	INT	NOT NULL	상품의 재고 수량 (필수 입력)

주문 (orders) 테이블

고객의 상품 주문 내역을 기록한다.

컬럼명	데이터 타입	제약 조건	설명
order_id	BIGINT	PK, AUTO_INCREMENT	주문의 고유 식별자 (자동 증가)
user_id	BIGINT	NOT NULL, FK	주문한 고객의 ID (users 테이블의 user_id 참조)
product_id	BIGINT	NOT NULL, FK	주문된 상품의 ID (products 테이블의 product_id 참조)
order_date	DATETIME	DEFAULT CURRENT_TIMESTAMP	주문 생성 일시 (자동으로 현재 시간 기록)
quantity	INT	NOT NULL	주문 수량 (필수 입력)
status	VARCHAR(50)	DEFAULT 'PENDING'	주문 상태 PENDING (대기) COMPLETED (완료) SHIPPED (배송) CANCELLED (취소)

직원 (employees) 테이블

직원 및 관리자(매니저) 관계를 정의한다. (Self Join 관계)

컬럼명	데이터 타입	제약 조건	설명

employee_id	BIGINT	PK, AUTO_INCREMENT	직원의 고유 식별자 (자동 증가)
name	VARCHAR(255)	NOT NULL	직원의 이름 (필수 입력)
manager_id	BIGINT	FK	해당 직원의 관리자 ID (employees 테이블의 employee_id를 참조). 최상위 관리자는 NULL 값을 가질 수 있음.

사이즈 (sizes) 및 색상 (colors) 테이블
상품의 다양한 옵션을 조합하기 위한 기준 데이터를 정의한다. (CROSS JOIN 실습용)

sizes 테이블

컬럼명	데이터 타입	제약 조건	설명
size	VARCHAR(10)	PK	상품의 사이즈 옵션 (e.g., 'S', 'M', 'L', 'XL')

colors 테이블

컬럼명	데이터 타입	제약 조건	설명
color	VARCHAR(20)	PK	상품의 색상 옵션 (e.g., 'Red', 'Blue', 'Black')

테이블 설계 및 생성

우리는 쇼핑몰의 핵심 데이터인 고객(users), 상품(products), 그리고 주문(orders)을 관리할 세 개의 기본 테이블을 설계했다. 또한, 이후의 다양한 조인 기법을 실습하기 위해 직원(employees) 테이블과 상품 옵션을 위한 sizes, colors 테이블도 설계했다. 이제 해당 테이블들을 실제로 만들어보자.

각 테이블을 생성하는 SQL(DDL)은 다음과 같다. 각 컬럼의 데이터 타입과 제약 조건을 유심히 살펴보길 바란다. 예를 들어, orders 테이블의 user_id는 users 테이블의 user_id를 참조하는 외래 키(Foreign Key)로 설정되어 데이터의 무결성을 지키도록 설계했다.

데이터베이스 이름의 경우 데이터베이스 입문 강의 예제와 겹치지 않게 하기 위해 my_shop2로 진행한다. 이 부분에

유의하자.

> **SQL 소스 파일 참고**
> 강의 자료가 PDF 파일이라 복잡한 SQL 코드를 복사할 때 오류가 발생할 수 있다.
> 이 경우, 섹션 1. 강의 소개와 수업 자료 ➜ SQL 소스 파일을 다운로드해서 사용하자.

```sql
-- 데이터베이스가 존재하지 않으면 생성
CREATE DATABASE IF NOT EXISTS my_shop2;
USE my_shop2;

-- 테이블이 존재하면 삭제 (실습을 위해 초기화)
DROP TABLE IF EXISTS orders;
DROP TABLE IF EXISTS users;
DROP TABLE IF EXISTS products;
DROP TABLE IF EXISTS employees;
DROP TABLE IF EXISTS sizes;
DROP TABLE IF EXISTS colors;

-- 고객 테이블 생성
CREATE TABLE users (
    user_id BIGINT AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    email VARCHAR(255) NOT NULL UNIQUE,
    address VARCHAR(255),
    birth_date DATE,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (user_id)
);

-- 상품 테이블 생성
CREATE TABLE products (
    product_id BIGINT AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    category VARCHAR(100),
    price INT NOT NULL,
    stock_quantity INT NOT NULL,
    PRIMARY KEY (product_id)
);

-- 주문 테이블 생성
```

```
CREATE TABLE orders (
    order_id BIGINT AUTO_INCREMENT,
    user_id BIGINT NOT NULL,
    product_id BIGINT NOT NULL,
    order_date DATETIME DEFAULT CURRENT_TIMESTAMP,
    quantity INT NOT NULL,
    status VARCHAR(50) DEFAULT 'PENDING',
    PRIMARY KEY (order_id),

    CONSTRAINT fk_orders_users FOREIGN KEY (user_id)
        REFERENCES users(user_id),

    CONSTRAINT fk_orders_products FOREIGN KEY (product_id)
        REFERENCES products(product_id)
);

-- 직원 테이블 생성 (SELF JOIN 실습용)
CREATE TABLE employees (
    employee_id BIGINT AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    manager_id BIGINT,
    PRIMARY KEY (employee_id),
    FOREIGN KEY (manager_id) REFERENCES employees(employee_id)
);

-- 사이즈 테이블 (CROSS JOIN 실습용)
CREATE TABLE sizes (
    size VARCHAR(10) PRIMARY KEY
);

-- 색상 테이블 (CROSS JOIN 실습용)
CREATE TABLE colors (
    color VARCHAR(20) PRIMARY KEY
);
```

- `CREATE DATABASE IF NOT EXISTS my_shop2`: `my_shop2` 데이터베이스가 존재하지 않으면 생성한다.
- `DROP TABLE IF EXISTS orders`: 주문 테이블이 만약에 존재하면 DROP 한다.
- 이렇게 `IF EXISTS` 구문을 활용하면 기존에 테이블이 있는 경우에만 깔끔하게 제거할 수 있다. 덕분에 같은 구문을 여러 번 실행해도 오류가 발생하지 않는다.
    - 만약 테이블이 없다면 DROP을 할 수 없기 때문에 DROP TABLE에서 오류가 발생한다. `IF EXISTS` 구

문은 이런 번거로움을 해결해준다.

샘플 데이터 입력

앞서 생성한 테이블에 앞으로 우리가 분석하고 다룰 샘플 데이터를 입력한다. 일부러 한 번도 주문하지 않은 고객(레오나르도 다빈치)과 한 번도 팔리지 않은 상품(고급 가죽 지갑)을 포함시켰다. 그 이유는 `OUTER JOIN` 수업에서 알게 될 것이다.

아래 `INSERT` 문을 실행해서 데이터를 채워 넣자.

```
-- 고객 데이터 입력
INSERT INTO users(name, email, address, birth_date) VALUES
('션', 'sean@example.com', '서울시 강남구', '1990-01-15'),
('네이트', 'nate@example.com', '경기도 성남시', '1988-05-22'),
('세종대왕', 'sejong@example.com', '서울시 종로구', '1397-05-15'),
('이순신', 'sunsin@example.com', '전라남도 여수시', '1545-04-28'),
('마리 퀴리', 'marie@example.com', '서울시 강남구', '1867-11-07'),
('레오나르도 다빈치', 'vinci@example.com', '이탈리아 피렌체', '1452-04-15');

-- 상품 데이터 입력
INSERT INTO products(name, category, price, stock_quantity) VALUES
('프리미엄 게이밍 마우스', '전자기기', 75000, 50),
('기계식 키보드', '전자기기', 120000, 30),
('4K UHD 모니터', '전자기기', 350000, 20),
('관계형 데이터베이스 입문', '도서', 28000, 100),
('고급 가죽 지갑', '패션', 150000, 15),
('스마트 워치', '전자기기', 280000, 40);

-- 주문 데이터 입력
INSERT INTO orders(user_id, product_id, quantity, status, order_date) VALUES
(1, 1, 1, 'COMPLETED', '2025-06-10 10:00:00'),
(1, 4, 2, 'COMPLETED', '2025-06-10 10:05:00'),
(2, 2, 1, 'SHIPPED', '2025-06-11 14:20:00'),
(3, 4, 1, 'COMPLETED', '2025-06-12 09:00:00'),
(4, 3, 1, 'PENDING', '2025-06-15 11:30:00'),
(5, 1, 1, 'COMPLETED', '2025-06-16 18:00:00'),
(2, 1, 2, 'SHIPPED', '2025-06-17 12:00:00');

-- 직원 데이터 입력
INSERT INTO employees(employee_id, name, manager_id) VALUES
(1, '김회장', NULL),
```

```
(2, '박사장', 1),
(3, '이부장', 2),
(4, '최과장', 3),
(5, '정대리', 4),
(6, '홍사원', 4);

-- 사이즈 데이터 입력
INSERT INTO sizes(size) VALUES
('S'), ('M'), ('L'), ('XL');

-- 색상 데이터 입력
INSERT INTO colors(color) VALUES
('Red'), ('Blue'), ('Black');
```

준비된 데이터 확인

모든 준비가 끝났다. 각 테이블에 데이터가 어떻게 들어갔는지 `SELECT` 문으로 직접 확인해 보자. 앞으로 우리는 이 데이터를 기반으로 흩어진 정보를 연결하고, 숨겨진 의미를 찾아내는 여정을 떠날 것이다.

`users` 테이블

```
SELECT * FROM users;
```

user_id	name	email	address	birth_date	created_at
1	션	sean@example.com	서울시 강남구	1990-01-15	(생성일시)
2	네이트	nate@example.com	경기도 성남시	1988-05-22	(생성일시)
3	세종대왕	sejong@example.com	서울시 종로구	1397-05-15	(생성일시)
4	이순신	sunsin@example.com	전라남도 여수시	1545-04-28	(생성일시)
5	마리 퀴리	marie@example.com	서울시 강남구	1867-11-07	(생성일시)

6	레오나르도 다빈치	vinci@example.com	이탈리아 피렌체	1452-04-15	(생성일시)

- `created_at` 필드(컬럼)의 날짜는 생성한 날짜이므로 각각 다르다.

`products` 테이블

```
SELECT * FROM products;
```

product_id	name	category	price	stock_quantity
1	프리미엄 게이밍 마우스	전자기기	75000	50
2	기계식 키보드	전자기기	120000	30
3	4K UHD 모니터	전자기기	350000	20
4	관계형 데이터베이스 입문	도서	28000	100
5	고급 가죽 지갑	패션	150000	15
6	스마트 워치	전자기기	280000	40

`orders` 테이블

```
SELECT * FROM orders;
```

order_id	user_id	product_id	order_date	quantity	status
1	1	1	2025-06-10 10:00:00	1	COMPLETED
2	1	4	2025-06-10 10:05:00	2	COMPLETED
3	2	2	2025-06-11 14:20:00	1	SHIPPED
4	3	4	2025-06-12 09:00:00	1	COMPLETED
5	4	3	2025-06-15 11:30:00	1	PENDING
6	5	1	2025-06-16 18:00:00	1	COMPLETED

7	2	1	2025-06-17 12:00:00	2	SHIPPED

이제 모든 준비가 완료되었다. 다음 시간부터 이 데이터를 가지고 `JOIN`의 세계로 본격적으로 뛰어들어 보자.

> **참고**
> 실무에서 다루는 주문 테이블들은 더 복잡하다. 지금은 데이터베이스 설계를 배우는 것이 목적이 아니기 때문에, 예제를 쉽게 이해할 수 있도록 테이블의 수를 최소화 했다. 실무에 가까운 복잡한 설계 예시는 데이터베이스 설계 강의에서 다루겠다.

조인이 필요한 이유

대표님이 "최근 주문 현황을 고객 이름과 상품명을 포함해서 보고서로 만들어줘!"라고 요청했다고 가정하자.

우리는 앞서 쇼핑몰 운영에 필요한 `users`, `products`, `orders` 테이블을 만들고 데이터를 입력했다.
주문 현황이 필요하기 때문에 `orders`의 정보를 제공하면 될 것 같다.

`orders` 테이블을 자세히 들여다보자.

```
SELECT * FROM orders;
```

> **my_shop2 데이터베이스 선택**
> `use my_shop2;`를 사용해서 `my_shop2` 데이터베이스를 선택하자.

[실행 결과]

order_id	user_id	product_id	order_date	quantity	status
1	1	1	2025-06-10 10:00:00	1	COMPLETED
2	1	4	2025-06-10 10:05:00	2	COMPLETED

3	2	2	2025-06-11 14:20:00	1	SHIPPED
4	3	4	2025-06-12 09:00:00	1	COMPLETED
5	4	3	2025-06-15 11:30:00	1	PENDING
6	5	1	2025-06-16 18:00:00	1	COMPLETED
7	2	1	2025-06-17 12:00:00	2	SHIPPED

대표님의 요구사항은 "최근 주문 현황을 고객 이름과 상품명을 포함해서 보고서로 만들어줘!"라는 것이었다. 그런데 `orders` 테이블에는 **고객 이름**과 **상품명**이 없다. 이 테이블만 보고서는 `user_id`가 1인 고객이 누구인지, `product_id`가 4인 상품이 무엇인지 즉시 알 수가 없다.

결국 다음과 같은 추가 과정이 필요하다.

- 고객 이름을 구하기 위해서는 `users` 테이블을 통해 `user_id`에 해당하는 고객명을 하나하나 직접 찾아야 한다.
- 상품명을 구하기 위해서는 `products` 테이블을 통해 `product_id`에 해당하는 상품명을 하나하나 직접 찾아야 한다.

지금처럼 데이터가 많이 없다면 고객명과 상품명을 각 테이블에서 직접 찾아서 보고서를 작성해도 되겠지만, 데이터가 수 백만 건이라면 상당히 큰 어려움이 있을 것이다.

왜 우리는 이렇게 불편하게 데이터를 여러 테이블에 나누어 저장하는 걸까? 차라리 처음부터 하나의 큰 테이블에 주문 정보, 고객 정보, 상품 정보를 모두 담아두면 편하지 않았을까?

만약 모든 데이터를 하나의 테이블에 저장한다면?

궁금증을 해결하기 위해, `users`, `products`, `orders` 테이블을 모두 제거하고, 이 테이블의 필드를 하나로 합친 `all_in_one` 테이블을 상상해 보자.

[상상 속의 거대한 테이블]

order_id	order_date	user_name	user_email	product_name	price	qua
1	2025-06-10	션	sean@...	프리미엄 게이밍 마우스	75000	1

2	2025-06-10	션	sean@...	관계형 데이터베이스 입문	28000	2
3	2025-06-11	네이트	nate@...	기계식 키보드	120000	1
...	...	...	...	...	...	...

- 참고용이므로 일부 필드는 제거했다.

당장은 편해 보인다. 하지만 이런 방식은 실무에서 재앙을 불러온다. 왜냐하면 다음과 같은 심각한 문제들이 발생하기 때문이다.

1. **데이터 중복 (Redundancy)**
   '션' 고객이 상품을 100번 주문했다고 생각해 보자. 그의 이름, 이메일, 주소 정보가 100번이나 불필요하게 반복 저장된다. 이건 매우 큰 저장 공간의 낭비다.

2. **갱신 이상 (Update Anomaly)**
   만약 '션' 고객이 이메일 주소를 변경했다고 가정해 보자. 우리는 '션'이 주문한 100개의 주문 데이터를 모두 찾아서, 이메일 정보를 일일이 새로운 주소로 변경해야 한다. 만약 실수로 단 하나라도 누락한다면? 어떤 주문에서는 고객의 이메일이 예전 주소로, 다른 주문에서는 새 주소로 저장되어 데이터의 일관성이 깨져버린다. 어떤 정보가 진짜인지 믿을 수 없게 되는 것이다.

3. **삽입 이상 (Insertion Anomaly)**
   우리 쇼핑몰에 아직 아무도 주문하지 않은 새로운 상품 '초경량 노트북'을 등록하고 싶다. 하지만 이 테이블 구조에서는 '주문'이 발생해야만 데이터를 추가할 수 있다. 주문한 사람이 없으니, 상품 정보조차 등록할 수 없는 말도 안 되는 상황이 발생한다.

4. **삭제 이상 (Deletion Anomaly)**
   '이순신' 고객이 딱 한 번 주문한 기록이 있다고 하자. 만약 회사 정책상 이 주문 기록을 삭제해야 한다면 어떻게 될까? 주문 데이터를 삭제하는 순간, '이순신' 고객의 이름, 이메일, 주소 정보까지 데이터베이스에서 영원히 사라져 버릴 수 있다. 우리는 단지 주문 내역 하나를 지웠을 뿐인데, 소중한 고객 정보까지 잃게 되는 것이다.

이러한 문제들 때문에 우리는 데이터베이스를 설계할 때 **정규화(Normalization)**라는 과정을 거친다. 정규화는 데이터의 중복을 최소화하고, 데이터의 일관성을 해치는 '이상 현상'들을 방지하기 위해 데이터를 논리적인 단위로 분리하는 과정이다. 우리가 `users`, `products`, `orders`로 테이블을 나눈 것이 바로 이 정규화의 결과물이다.

> 정규화

> 정규화는 데이터베이스 설계의 기본을 이루는 중요한 이론이다.
> 정규화에 대한 자세한 내용은 데이터베이스 설계 강의에서 다룬다.
> 지금은 이런 문제들 때문에 여러 테이블에 나누어 저장한다는 것 정도만 대략 이해하면 충분하다.

그래서 조인이 필요하다

이제 우리는 왜 데이터를 분리해서 저장하는지 이해했다. 데이터의 중복을 막고, 일관성을 지키기 위해서다. 즉, 데이터를 '잘 관리하기 위해서'다. 하지만 잘 관리하기 위해 흩어놓은 데이터에서 의미 있는 정보를 얻으려면, 이 흩어진 조각들을 다시 합쳐야만 한다. "어떤 고객이 어떤 상품을 주문했는지"와 같은 통합된 보고서를 만들기 위해, 분리된 테이블들을 다시 연결해야 하는 것이다.

이때 사용하는 기술이 바로 조인(`JOIN`)이다.

조인은 두 개 이상의 테이블을 특정 컬럼을 기준으로 연결하여, 마치 처음부터 하나의 테이블이었던 것처럼 보여주는 기능이다. 보통 테이블을 설계할 때 연결고리로 사용하는 `기본 키(Primary Key)`와 `외래 키(Foreign Key)`를 사용해 이들을 합친다.

- `orders` 테이블의 `user_id` 외래 키(FK)는 `users` 테이블의 `user_id` 기본 키(PK)와 연결된다.
- `orders` 테이블의 `product_id` 외래 키(FK)는 `products` 테이블의 `product_id` 기본 키(PK)와 연결된다.

조인은 데이터 정규화(분리)를 통해 얻는 일관성과 효율성의 장점은 그대로 유지하면서, 우리가 원하는 통합된 정보를 얻을 수 있게 해주는, 데이터 분리와 통합을 완성하는 기술이다.

이제 왜 조인을 배워야 하는지 대략 이해했을 것이다. 다음 시간부터는 가장 기본적이고 중요한 내부 조인(`INNER JOIN`)을 시작으로, 흩어진 우리 쇼핑몰의 데이터를 연결하여 의미 있는 보고서를 만드는 실습을 본격적으로 진행하겠다.

내부 조인 1

지난 시간에 우리는 데이터의 일관성과 효율성을 위해 테이블을 분리(정규화)하며, 이렇게 분리된 테이블을 다시 합쳐 의미 있는 정보를 얻기 위해 조인(`JOIN`)이 필요하다는 사실을 깨달았다.

우리 쇼핑몰의 대표님이 다음과 같은 요청을 한다.
"주문이 완료된(`COMPLETED`) 모든 주문에 대해, 어떤 고객이 주문했는지 고객 ID, 고객 이름과 주문 날짜를 함께 보고 싶네."

이 요구사항을 충족하려면 '고객 이름'을 담고 있는 `users` 테이블과, '주문 날짜'와 '주문 상태'를 담고 있는 `orders` 테이블의 정보가 모두 필요하다. 즉, 두 테이블을 연결해야만 한다.

조인은 크게 내부 조인(`INNER JOIN`)과 외부 조인(`OUTER JOIN`)으로 나눌 수 있다.
우선 내부 조인부터 자세히 알아보자.

내부 조인의 개념

내부 조인(`INNER JOIN`)은 두 테이블을 연결할 때, **양쪽 테이블에 모두 공통으로 존재하는 데이터만**을 결과로 보여준다.
기준이 되는 컬럼(예: `orders.user_id`와 `users.user_id`)의 값이 서로 일치하는 행들만 짝을 지어주는 것이다.

비유하자면 소개팅 앱에서 A와 B가 서로 '좋아요'를 눌렀을 때만 연결이 성사되는 것과 같다.
`orders` 테이블에 `user_id`가 존재하고, 연결되는 `users` 테이블에도 해당 `user_id`를 가진 사용자가 존재할 때만 결과에 포함된다.

- 예를 들어 `orders` 테이블에 `user_id`가 1인 주문이 있고 `users` 테이블에 `user_id`가 1인 회원이 있다면 둘은 함께 결과에 포함된다.
- 예를 들어 `orders` 테이블에 `user_id`가 99인 주문이 있지만 `users` 테이블에 `user_id`가 99인 회원이 없다면, 결과에서 제외된다. (실제로는 데이터 무결성을 위해 외래 키(FOREIGN KEY) 제약조건을 사용하므로 이런 경우는 발생하지 않는다.)

내부 조인 문법

```sql
SELECT 컬럼1, 컬럼2, ...
FROM 테이블A
INNER JOIN 테이블B
ON 테이블A.연결컬럼 = 테이블B.연결컬럼;
```

- `FROM`: 기준이 되는 첫 번째 테이블을 지정한다.
- `INNER JOIN`: 연결할 두 번째 테이블을 지정한다.
- `ON`: **조인에서 가장 중요한 부분이다.** 두 테이블을 어떤 조건으로 연결할지 명시하는 연결고리다. `ON` 절의 조건이 참(true)이 되는 행들만 결과에 포함된다.

> **INNER 생략**
> 내부 조인에서 `INNER`는 생략할 수 있다. 그냥 `JOIN`이라고만 쓰면 `INNER JOIN`으로 작동한다.
> 내부 조인을 사용하는 경우 실무에서는 대부분 `INNER`를 생략하고 `JOIN`만 사용한다.

실습: 주문별 고객 정보 조회하기

실제 쿼리를 작성하며 내부 조인(`INNER JOIN`)을 자세히 알아보자.

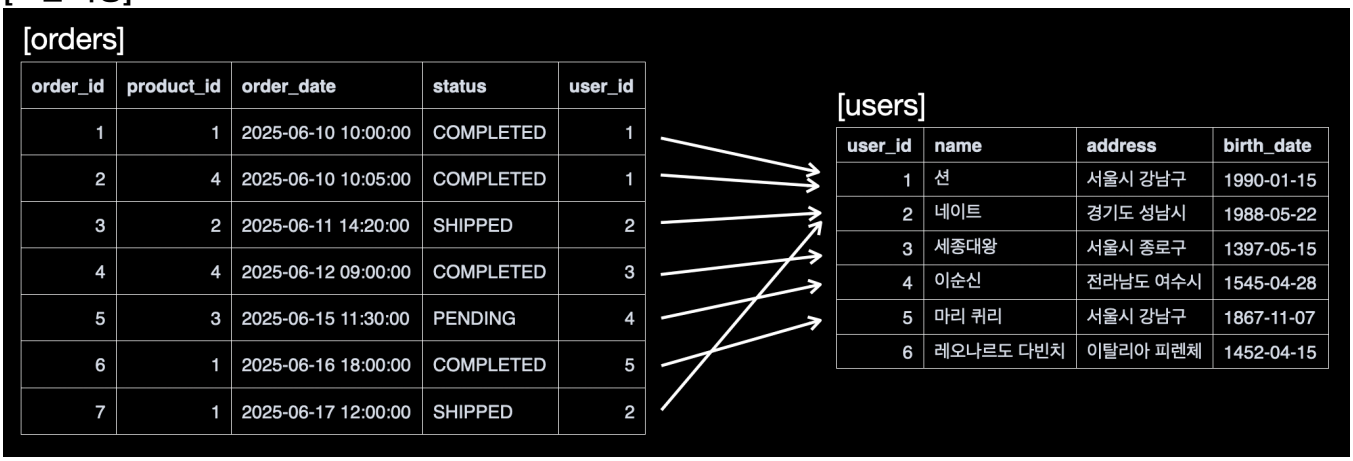
1단계: 두 테이블을 그대로 연결하기

먼저 `orders` 테이블과 `users` 테이블을 `orders.user_id`와 `users.user_id`를 기준으로 연결해 보자.
`SELECT *`를 사용해서 어떤 결과가 나오는지 눈으로 직접 확인하자.

```sql
SELECT *
FROM orders
INNER JOIN users ON orders.user_id = users.user_id;
```

실행 결과는 `orders` 테이블의 모든 컬럼과 `users` 테이블의 `user_id`가 같은 모든 컬럼이 옆으로 합쳐진, 아주 넓은 테이블이 된다.
조인은 쉽게 이야기해서 테이블을 옆으로 합치는 것이다.

[조인 과정]



- 데이터베이스는 `orders` 테이블과 `users` 테이블을 조회한다.
- `JOIN ON`에 의해 `orders`의 `user_id`와 `users`의 `user_id`를 기준으로 조인한다.
- `orders`의 `user_id`의 값으로 `users`의 `user_id` 항목을 찾고 연결한다.
  - `order_id:1`의 경우 `user_id:1`이다. 따라서 `users`의 `user_id:1` 항목과 연결한다.
  - `order_id:2`의 경우 `user_id:1`이다. 따라서 `users`의 `user_id:1` 항목과 연결한다.
  - `order_id:3`의 경우 `user_id:2`이다. 따라서 `users`의 `user_id:2` 항목과 연결한다.

    - `order_id:4`의 경우 `user_id:3`이다. 따라서 `users`의 `user_id:3` 항목과 연결한다.
    - ...
    - `order_id:7`의 경우 `user_id:2`이다. 따라서 `users`의 `user_id:2` 항목과 연결한다.
- `user_id:6`인 레오나르도 다빈치의 경우 연결할 대상이 없다. 따라서 내부 조인 대상에 포함되지 않는다.

> 강의 전반에 걸쳐 설명을 단순화하기 위해 그림과 실행 결과에서 중요하지 않은 일부 필드는 제외하겠다.

[실행 결과]

order_id	user_id	product_id	order_date	quantity	status	user_id	name	email	address	birth_date	created_at
1	1	1	2025-06-10 10:00:00	1	COMPLETED	1	션	sean@example.com	서울시 강남구	1990-01-15	(생성일시)
2	1	4	2025-06-10 10:05:00	2	COMPLETED	1	션	sean@example.com	서울시 강남구	1990-01-15	(생성일시)
3	2	2	2025-06-11 14:20:00	1	SHIPPED	2	네이트	nate@example.com	경기도 성남시	1988-05-22	(생성일시)
4	3	4	2025-06-12 09:00:00	1	COMPLETED	3	세종대왕	sejong@example.com	서울시 종로구	1397-05-15	(생성일시)
5	4	3	2025-06-15 11:30:00	1	PENDING	4	이순신	sunsin@example.com	전라남도 여수시	1545-04-28	(생성일시)
6	5	1	2025-06-16 18:00:00	1	COMPLETED	5	마리 퀴리	marie@example.com	서울시 강남구	1867-11-07	(생성일시)
7	2	1	2025-06-17 12:00:00	2	SHIPPED	2	네이트	nate@example.com	경기도 성남시	1988-05-22	(생성일시)

- 결과에서 볼 수 있듯이, `orders`와 `users`의 모든 컬럼이 옆으로 합쳐진다.
- `user_id` 컬럼이 두 번 나타나는 이유는 `orders`, `users` 두 테이블 모두에 존재하기 때문이다.
- **참고**: SELECT에 `*` 대신 컬럼을 직접 나열하는 경우 `user_id` 처럼 중복으로 나타나는 컬럼은 테이블 명을 지정해주어야 한다.

2단계: 필요한 컬럼만 선택하고, `WHERE`로 필터링하기

`SELECT *`는 구조를 파악하는 데는 좋지만, 실제 보고서에는 불필요한 정보가 너무 많다. 이제 대표님의 요구사항에 맞게 '고객 ID', '고객 이름', '주문 날짜'만 선택하고, `WHERE` 절을 추가하여 주문이 완료된 `COMPLETED` 상태의 주문만 필터링해 보자.

```sql
SELECT
    users.user_id,    -- 테이블 명 생략 불가
    users.name,       -- 테이블 명 생략 가능
    orders.order_date -- 테이블 명 생략 가능
FROM orders
INNER JOIN users ON orders.user_id = users.user_id
WHERE orders.status = 'COMPLETED';
```

- 여기서 `users.user_id`는 `테이블명.컬럼명` 형식으로 작성한 것을 볼 수 있다.
- `user_id`처럼 `orders`, `users` 두 테이블에 이름이 같은 컬럼이 있을 경우, 어떤 테이블의 `user_id`를 말하는지 명확히 지정해주어야 한다. 그렇지 않으면 오류가 발생한다.
  - 오류 메시지: `Column 'user_id' in field list is ambiguous` (`user_id` 필드가 모호하다는 오류 메시지)
- `name`, `order_date`의 경우 한 테이블에만 존재하기 때문에 모호함이 발생하지는 않는다. 이 경우 앞의 테이블명을 생략할 수 있다.
- 실무에서는 어떤 테이블의 필드인지 명시하는 것이 가독성을 높이고, 컬럼의 소속을 쉽게 인지할 수 있기 때문에 사용하는 것을 권장한다. (참고로 뒤에서 설명할 테이블 별칭과 함께 사용하는 것이 좋다.)

[실행 결과]

user_id	name	order_date
1	션	2025-06-10 10:00:00
1	션	2025-06-10 10:05:00
3	세종대왕	2025-06-12 09:00:00
5	마리 퀴리	2025-06-16 18:00:00

드디어 대표님이 요구한 '결제 완료된 주문의 고객ID, 고객명과 주문 날짜' 보고서를 완성했다.

조인 작동 순서

결제 완료된 주문의 고객명과 주문 날짜가 어떤 순서로 조회된 것인지 조인이 작동하는 논리적인 순서를 알아보자.

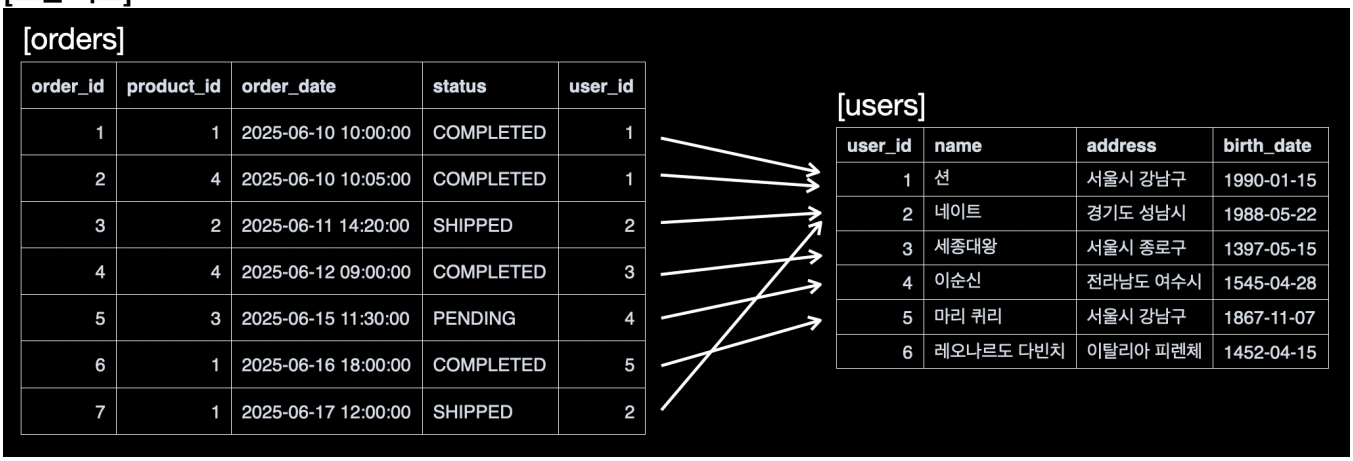
```
SELECT
    users.user_id,
    users.name,
    orders.order_date
FROM orders
INNER JOIN users ON orders.user_id = users.user_id
WHERE orders.status = 'COMPLETED';
```

데이터베이스는 이 쿼리를 다음과 같은 논리적인 순서로 처리한다.

1. `FROM` / `JOIN`: 가장 먼저 `FROM` 절의 `orders` 테이블과 `INNER JOIN`으로 연결된 `users` 테이블을 연결하기

위해 `ON` 절에 명시된 `orders.user_id = users.user_id` 조건을 만족하는 행들을 결합하여 하나의 큰 가상 테이블을 생성한다.

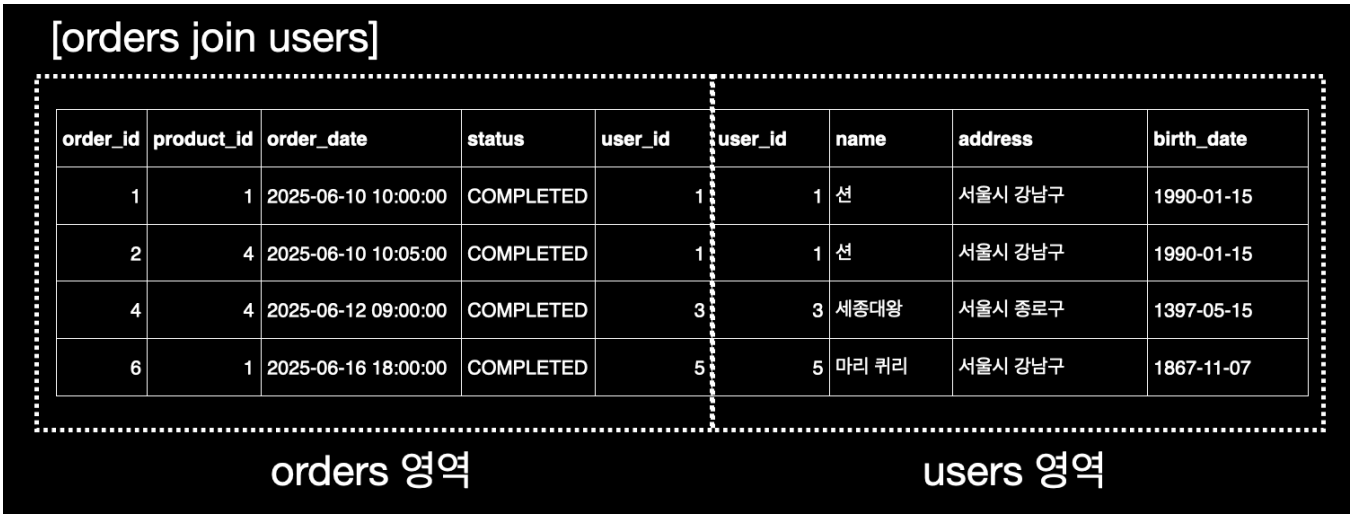
[조인 시도]



[조인 결과]

2. `WHERE` : `JOIN` 을 통해 생성된 가상 테이블에서 `WHERE` 절의 조건인 `orders.status = 'COMPLETED'` 를 만족하는 행들만 필터링한다. 즉, `COMPLETED` 상태의 주문 데이터만 남게 된다

[WHERE 필터링]



3. `SELECT`: 마지막으로, 필터링된 결과에서 `SELECT` 절에 명시된 `users.user_id`, `name`과 `order_date` 컬럼을 추출하여 최종 결과를 반환한다.

[최종 결과 - SELECT 반영]

user_id	name	order_date
1	션	2025-06-10 10:00:00
1	션	2025-06-10 10:05:00
3	세종대왕	2025-06-12 09:00:00
5	마리 퀴리	2025-06-16 18:00:00

순서를 정리해보자.
`JOIN`을 통해 두 테이블을 먼저 합친 가상의 테이블을 만든 후, `WHERE` 절의 조건에 따라 필요한 행을 걸러낸다. 그리고 최종적으로 원하는 필드를 `SELECT`로 선택하는 순서로 작동한다.

요약: 논리적 처리 순서

- `FROM/JOIN (테이블 결합)` → `WHERE (조건 필터링)` → `SELECT (컬럼 선택)`

> **쿼리 최적화기 (쿼리 옵티마이저)**
> 사용자와 데이터베이스 간의 논리적 순서는 일종의 약속이지만, 데이터베이스 내부의 **쿼리 최적화기 (Query Optimizer)**는 쿼리를 더 효율적인 방식으로 실행한다.
> 예를 들어, `orders` 테이블에서 `COMPLETED` 상태의 행만 먼저 선택한 다음, 남은 행을 기준으로 `users` 테이블과 조인하는 방식으로 진행하면 조인 대상이 줄어들어 성능이 더 최적화 될 수 있다.

> 쿼리 최적화기를 통해 실제 물리적인 실행 순서는 달라질 수 있지만 어떻게 작동하든 최종 결과는 논리적인 순서와 동일하다. 하지만 데이터베이스의 최적화 방식을 잘 이해하면 같은 결과를 얻으면서도 조회 성능을 최적화할 수 있다.
>
> 성능 최적화에 대한 자세한 내용은 데이터베이스 성능 최적화 강의에서 다룬다. 지금은 SQL의 기능 자체에 집중하자

내부 조인 2

조인과 집합

내부 조인을 이해하는 또 다른 방법은 바로 '집합'의 관점에서 바라보는 것이다. 학창 시절 수학 시간에 배운 벤 다이어그램을 떠올리면 아주 쉽게 이해할 수 있다.

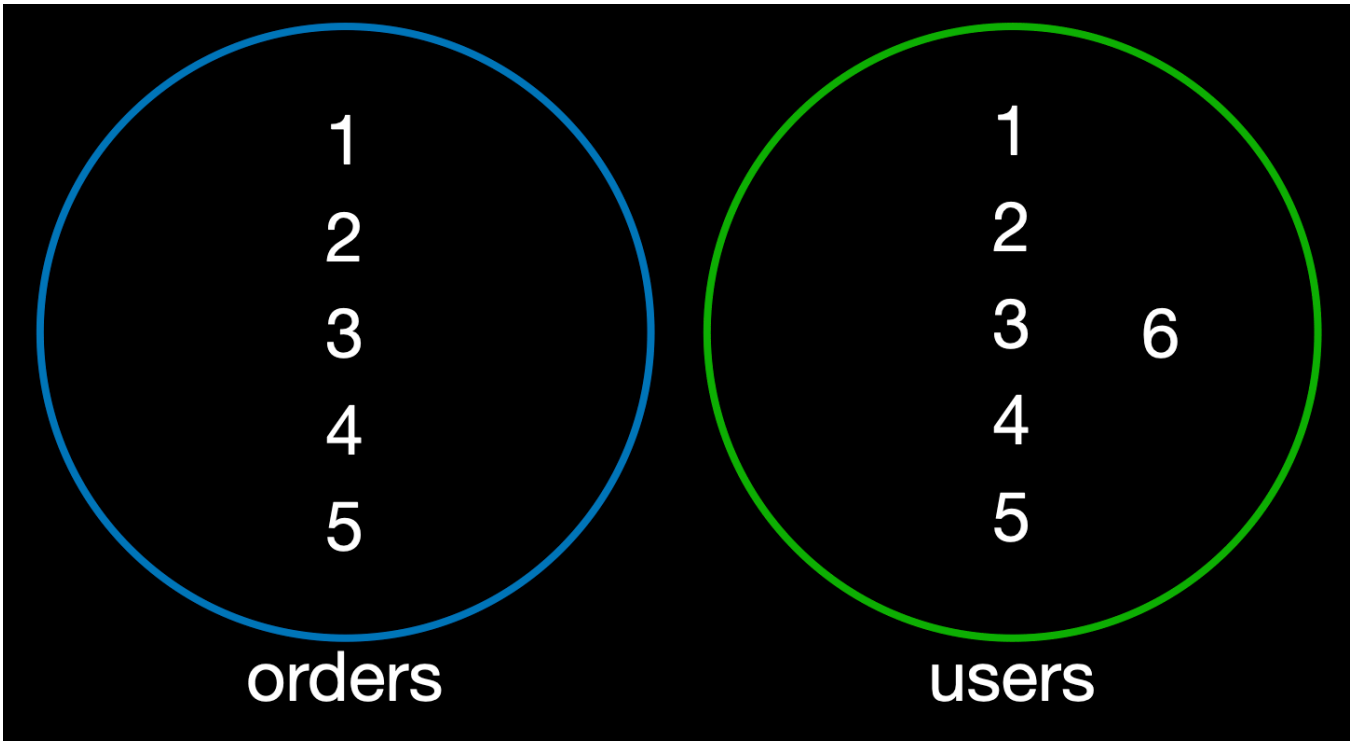
> **어려운 수학을 이야기 하는 것이 아니다! 단순한 동그라미 그리기 관점으로 이해하자!**

내부 조인(`INNER JOIN`)은 두 테이블의 **교집합**을 찾는 것과 같다.
두 집합(테이블)에서 공통된 원소(연결 컬럼의 값이 일치하는 데이터)만을 결과로 반환한다.

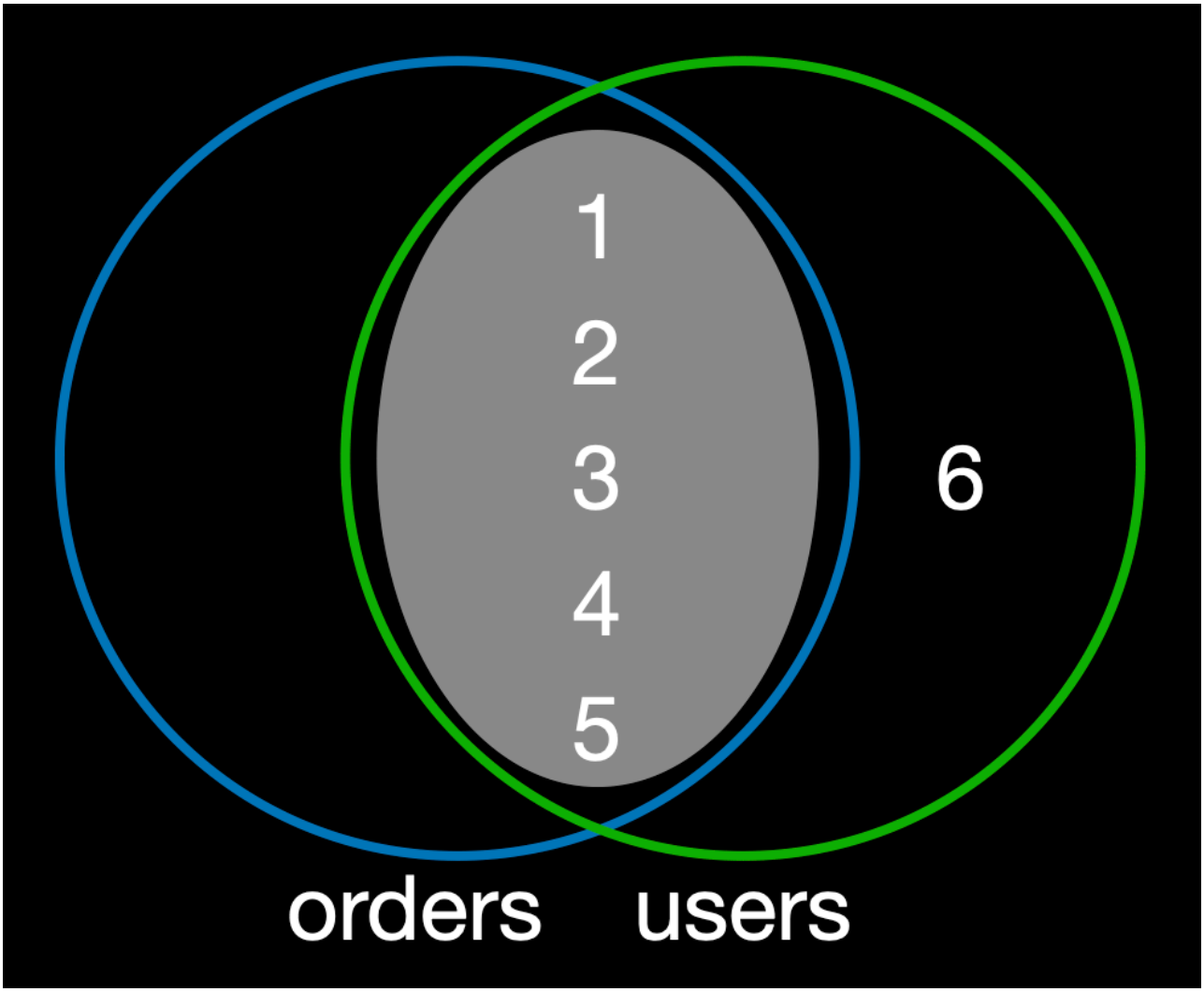
- **A 집합**: `orders` 테이블에 있는 모든 사용자들의 `user_id` 집합
- **B 집합**: `users` 테이블에 있는 모든 `user_id` 집합

`INNER JOIN`은 A 집합과 B 집합에 **모두 포함된** `user_id`에 해당하는 데이터만을 결합하여 보여준다.

벤 다이어그램으로 이해하기
우리의 데이터 상황을 벤 다이어그램으로 그려보면 다음과 같다.



- `orders` **집합 (왼쪽 원)**: 지금까지 들어온 모든 주문을 나타낸다. 주문 데이터에 기록된 `user_id`는 `{1, 2, 3, 4, 5}`가 있다. (참고로 집합은 중복 값은 제거하고 표현한다)
- `users` **집합 (오른쪽 원)**: 우리 쇼핑몰에 가입한 모든 회원을 나타낸다. `user_id`가 `{1, 2, 3, 4, 5, 6}`이 있다.

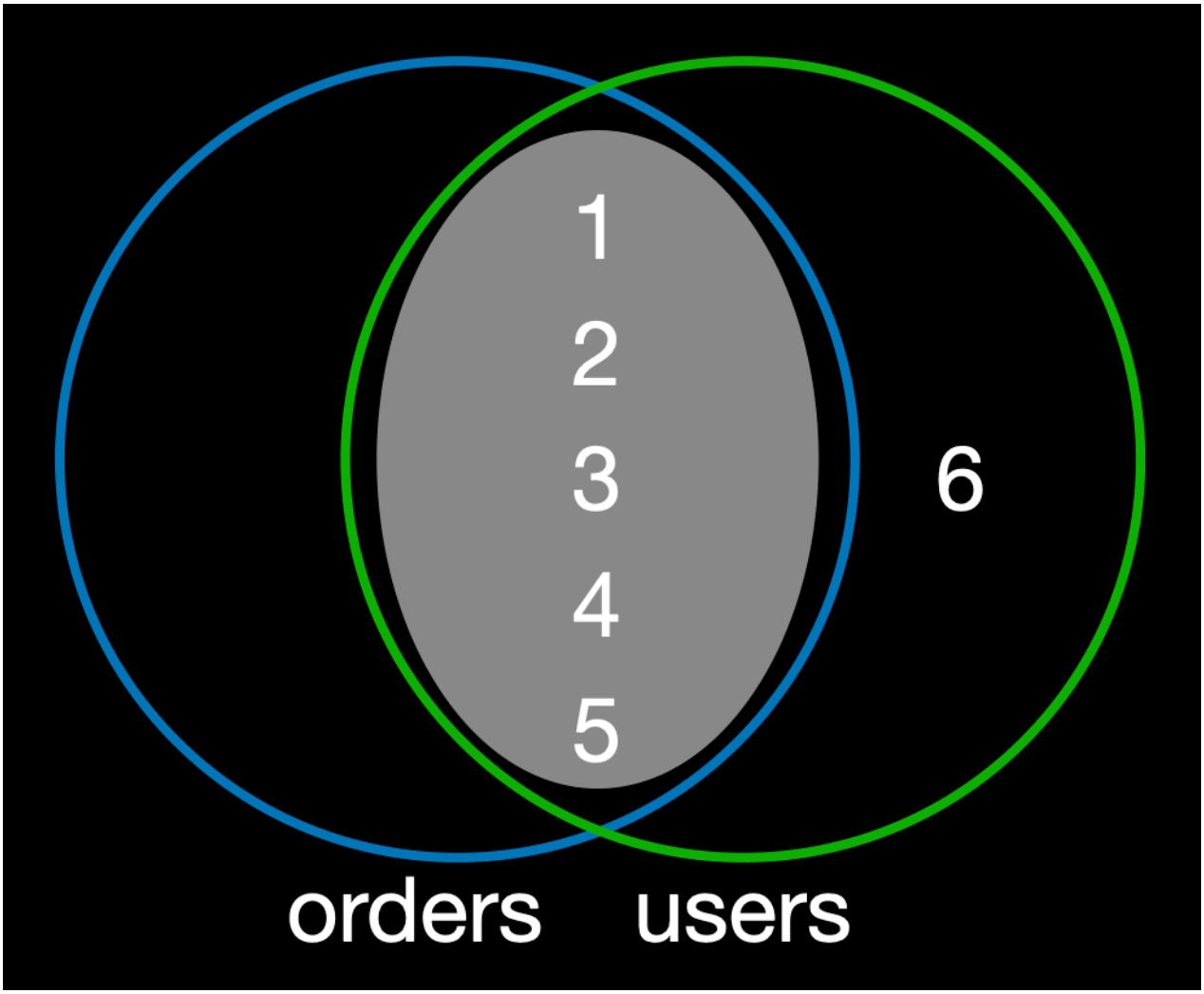


- **교집합 (가운데 영역)**: `users`와 `orders` 양쪽에 모두 존재하는 `user_id`의 집합 `{1, 2, 3, 4, 5}`이다. 내부 조인의 결과는 바로 이 교집합에 해당하는 데이터들이다.

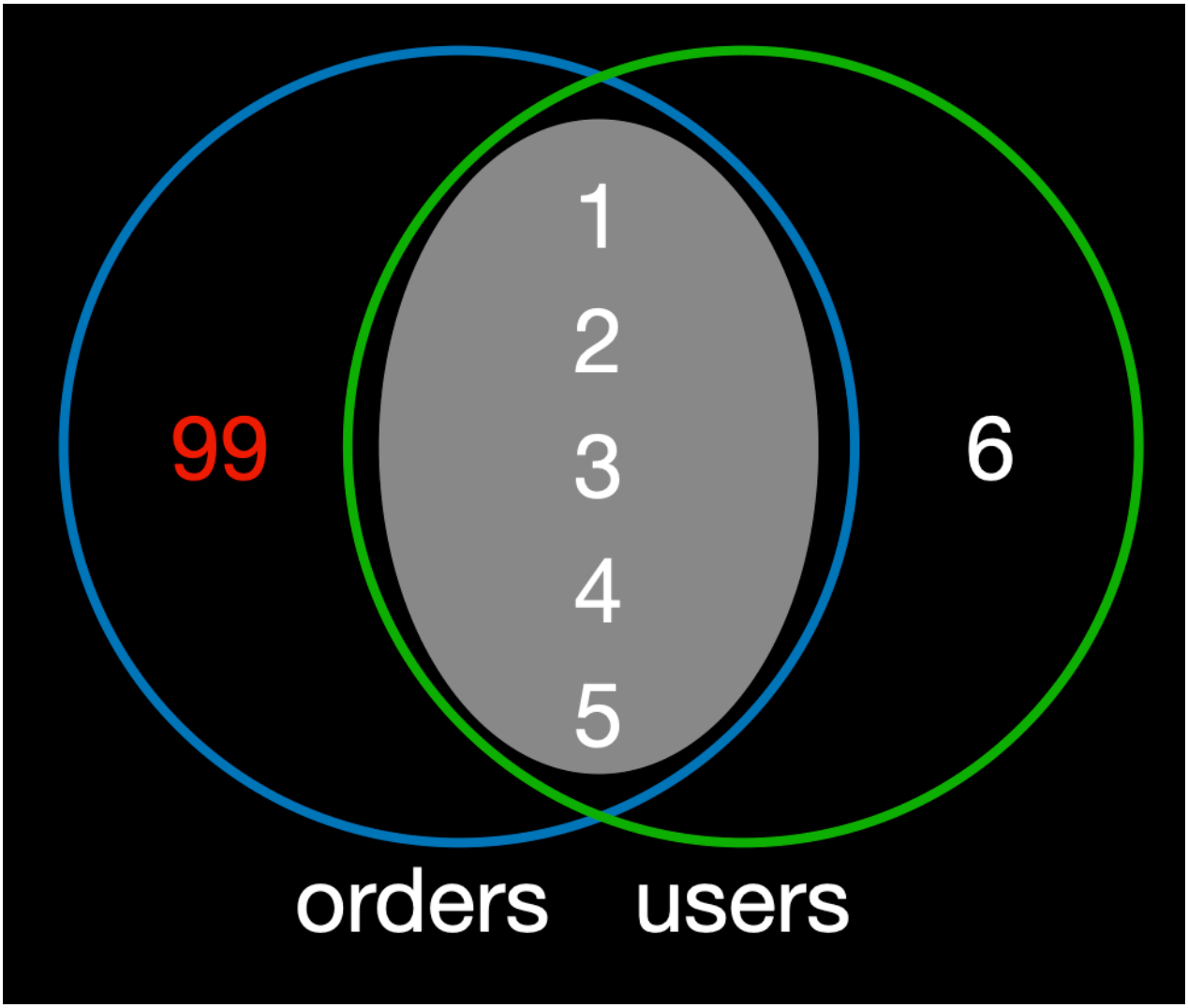
> **용어 - 내부 조인**
> 벤 다이어그램을 보면 내부 조인(`INNER JOIN`)의 이름을 왜 내부라고 지었는지 알 수 있다.
> 내부 조인은 벤 다이어그램에서 둘의 겹친 영역인 교집합 영역을 말한다.
> 교집합 영역은 벤 다이어그램에서 내부에 있는 데이터를 뜻한다.
> 이후에 설명할 외부 조인(`OUTER JOIN`)은 교집합 영역의 밖(`OUTER`)의 행까지 포함한다는 의미이다.

집합의 관점에서 제외되는 데이터

그렇다면 교집합에 포함되지 않는 데이터는 어떻게 될까?



1. **주문 기록이 없는 회원**: `user_id`가 6인 '레오나르도 다빈치'는 `users` 테이블에는 존재하지만, 아직 한 번도 주문한 적이 없으므로 `orders` 테이블에는 해당 `user_id`가 없다. 따라서 이 회원은 `users` 집합에는 속하지만 교집합에는 속하지 않으므로 `INNER JOIN` 결과에서 제외된다.



2. **존재하지 않는 회원의 주문**: 만약 `orders` 테이블에 `user_id`가 99인 주문이 있는데, `users` 테이블에 `user_id`가 99인 회원이 없다면 어떻게 될까? 이 주문 데이터는 `orders` 집합에는 속하지만 교집합에는 포함되지 않으므로 `INNER JOIN` 결과에서 제외된다. (실제로는 데이터 무결성을 위해 외래 키(FOREIGN KEY) 제약조건을 사용하므로 이런 경우는 발생하지 않는다.)

결론적으로, `INNER JOIN`은 어느 한쪽에만 데이터가 존재하는 경우는 결과에 포함시키지 않고, **양쪽 모두에 명확하게 연결고리가 있는 데이터만**을 짝지어 보여준다.

쿼리로 확인하기

내부 조인을 사용했을 때 교집합에 존재하는 `user_id` `{1,2,3,4,5}`만 선택되었는지 쿼리로 확인해보자.

```
SELECT
    orders.order_id,
    orders.order_date,
    orders.user_id AS orders_user_id,
    users.user_id AS users_user_id,
    users.name
FROM orders
```

```
INNER JOIN users ON orders.user_id = users.user_id
ORDER BY orders.order_id;
```

이 쿼리는 `orders` 테이블과 `users` 테이블의 `user_id`를 기준으로 교집합을 찾은 다음(`INNER JOIN`), 최종적으로 요청한 컬럼을 선택(`SELECT`)한 것이다.

[실행 결과]

order_id	order_date	orders_user_id	users_user_id	name
1	2025-06-10 10:00:00	1	1	션
2	2025-06-10 10:05:00	1	1	션
3	2025-06-11 14:20:00	2	2	네이트
4	2025-06-12 09:00:00	3	3	세종대왕
5	2025-06-15 11:30:00	4	4	이순신
6	2025-06-16 18:00:00	5	5	마리 퀴리
7	2025-06-17 12:00:00	2	2	네이트

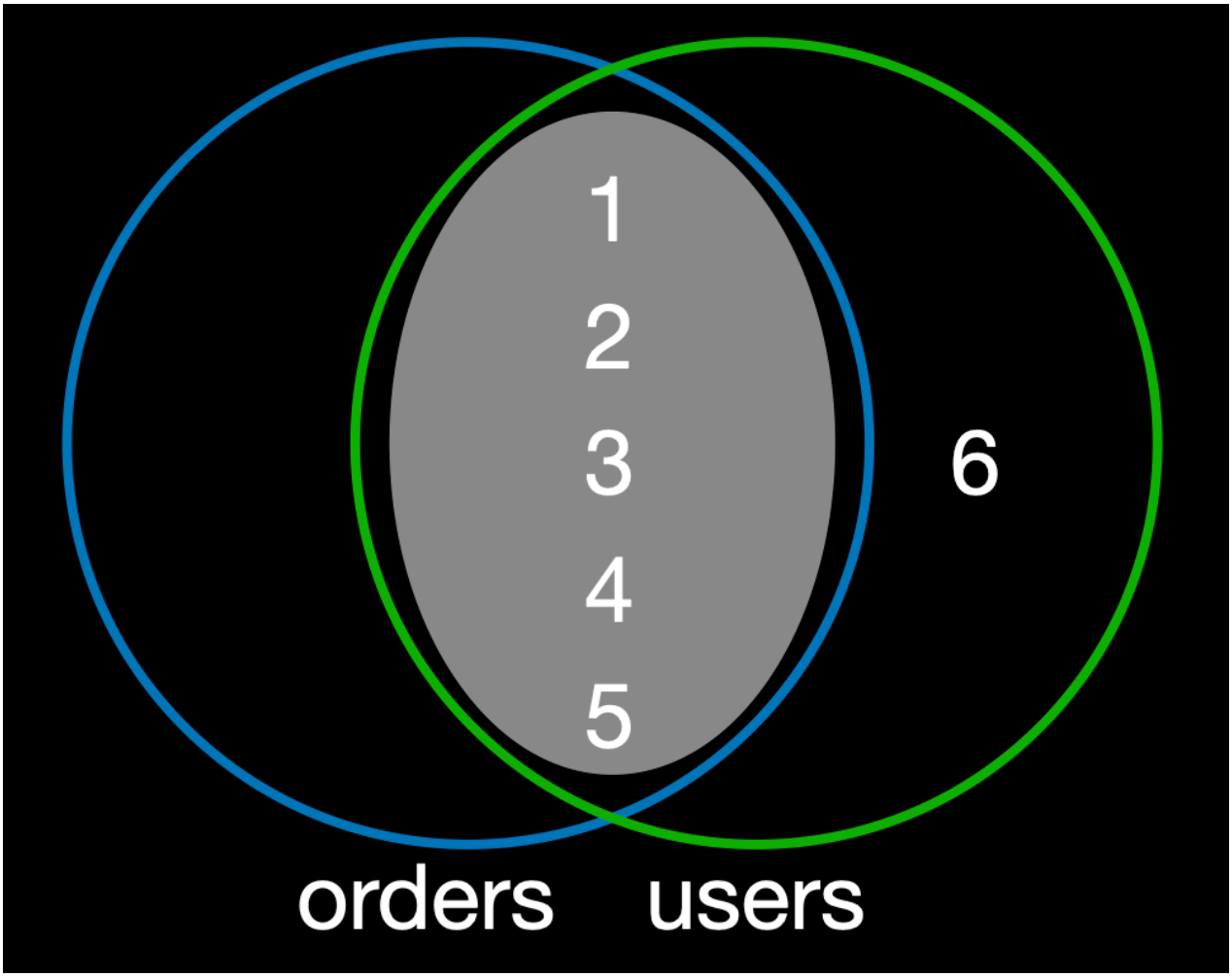
- `user_id`가 양쪽에 모두 있는 `{1,2,3,4,5}`가 선택되었다.
- 주문 기록이 없는 `user_id: 6` ('레오나르도 다빈치')는 포함되지 않은 것을 다시 한번 확인할 수 있다.

이처럼 조인을 집합의 관점에서 이해하면, 앞으로 배우게 될 다양한 종류의 외부 조인(`OUTER JOIN`)이 벤 다이어그램의 어떤 부분을 결과로 가져오는지 훨씬 직관적으로 파악할 수 있게 된다.

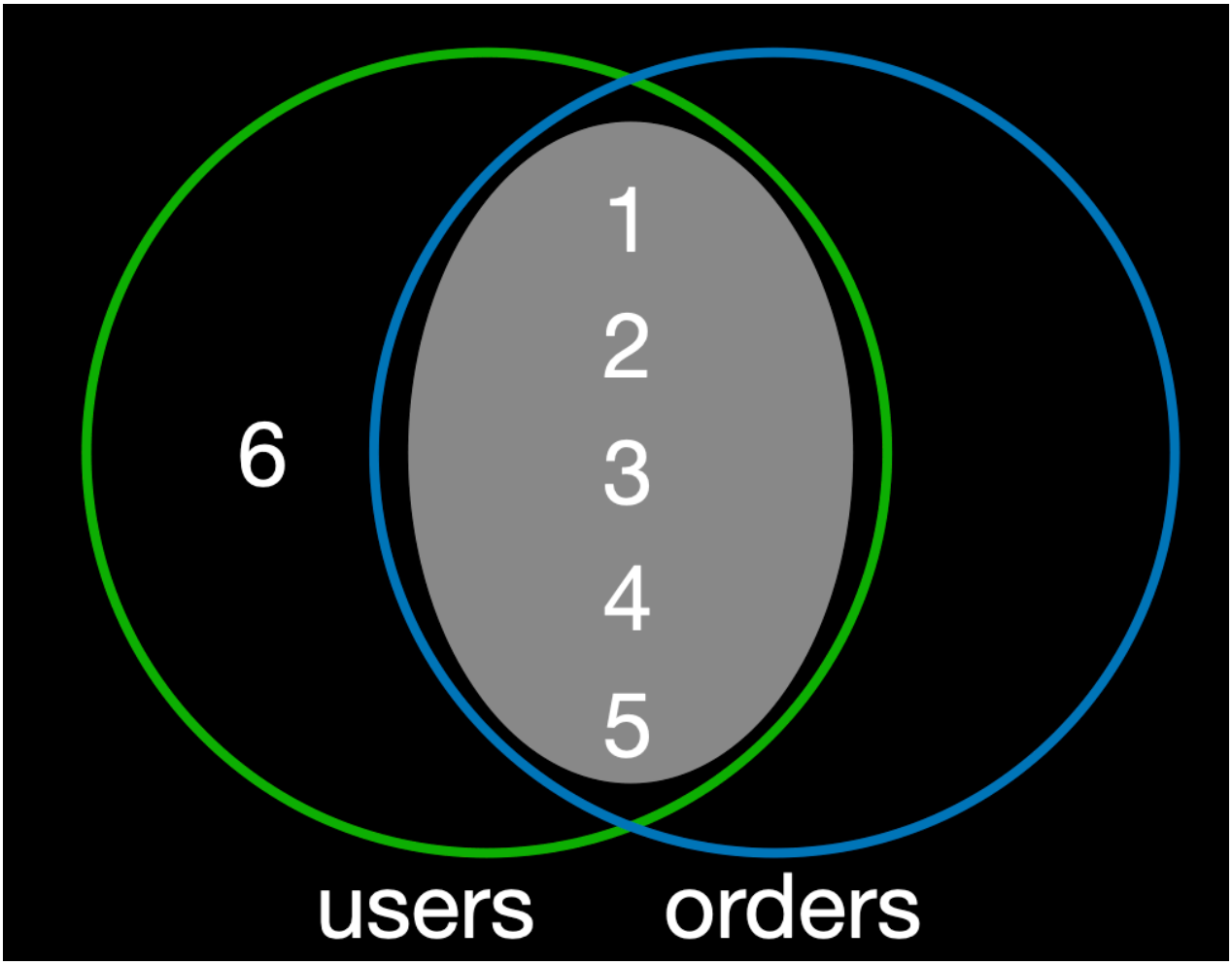
내부 조인과 조인 방향

내부 조인은 양방향이다. A 테이블과 B 테이블이 있다고 하면, A ➜ B로 조인할 수 있다면 반대로 B ➜ A로 조인할 수 있다.
그리고 **그 결과는 항상 동일**하다.

왜 결과가 동일할까?
내부 조인은 두 테이블 간의 **교집합**을 찾는 연산이기 때문이다.
`A와 B의 교집합`과 `B와 A의 교집합`이 같은 것과 같은 원리다.



- `orders`가 왼쪽, `users`가 오른쪽
- 교집합은 `{1,2,3,4,5}`이다.



- `users`가 왼쪽, `orders`가 오른쪽
- 교집합은 `{1,2,3,4,5}`이다.

이런 이유로 교집합을 확인하는 내부 조인은 A ➜ B로 조인하든 B ➜ A로 조인하든 결과가 항상 같다.

그림으로 확인하기

조금 더 구체적으로 확인해보자.

[`orders` ➜ `users` 조인]

[orders]

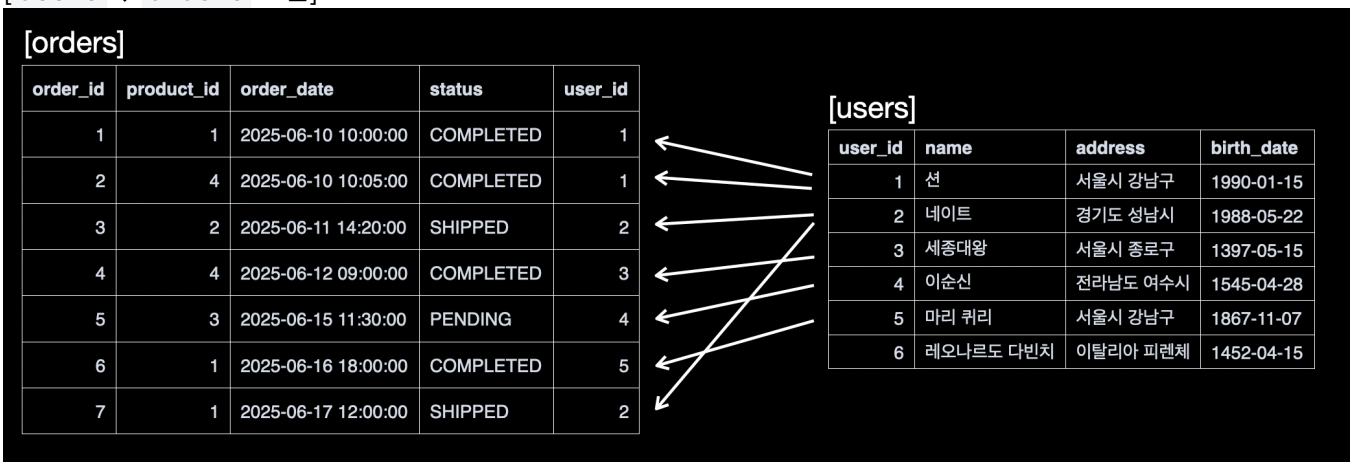
order_id	product_id	order_date	status	user_id
1	1	2025-06-10 10:00:00	COMPLETED	1
2	4	2025-06-10 10:05:00	COMPLETED	1
3	2	2025-06-11 14:20:00	SHIPPED	2
4	4	2025-06-12 09:00:00	COMPLETED	3
5	3	2025-06-15 11:30:00	PENDING	4
6	1	2025-06-16 18:00:00	COMPLETED	5
7	1	2025-06-17 12:00:00	SHIPPED	2

[users]

user_id	name	address	birth_date
1	션	서울시 강남구	1990-01-15
2	네이트	경기도 성남시	1988-05-22
3	세종대왕	서울시 종로구	1397-05-15
4	이순신	전라남도 여수시	1545-04-28
5	마리 퀴리	서울시 강남구	1867-11-07
6	레오나르도 다빈치	이탈리아 피렌체	1452-04-15

- 데이터베이스는 `orders` 테이블과 `users` 테이블을 조회한다.
- `orders` ➜ `users` 조인: `orders` 테이블을 한 줄씩 읽으며, 각 주문의 `user_id`에 해당하는 `users` 정보를 찾아 옆에 붙인다.

[`users` ➜ `orders` 조인]



- 데이터베이스는 `users` 테이블과 `orders` 테이블을 조회한다. 그림에서 화살표의 방향이 반대인 것을 확인하자.
- `users` ➜ `orders` 조인: `users` 테이블을 한 줄씩 읽으며, 각 사용자의 `user_id`와 일치하는 모든 `orders` 정보를 찾아 옆에 붙인다.
  - 예를 들어 `users`의 `user_id:1`은 `orders`의 `user_id:1`인 `order_id:1`, `order_id:2`와 연결된다.

두 방식 모두 `ON` 조건을 만족하는 모든 조합을 찾아내므로, 논리적으로 완전히 동일한 결과를 생성한다. 주문이 없는 '레오나르도 다빈치'나, 주문은 있지만 존재하지 않는 회원(만약 그런 데이터가 있다면)은 어느 방향으로 조인해도 결과에 포함되지 않는다.

쿼리로 확인하기

앞서 `orders`의 `user_id`를 사용해서 `users`에 있는 `user_id`로 조인했다. 쉽게 이야기해서 `orders` ➜ `users` 방향으로 조인했다.
그럼 반대로 `users`의 `user_id`를 사용해서 `orders`에 있는 `user_id`로 조인하면 어떻게 될까? 쉽게 이야기해서 반대 방향인 `users` ➜ `orders` 방향으로 조인하면 어떻게 될까?

직접 쿼리를 실행해서 확인해 보자.

```
SELECT
    orders.order_id,
    orders.order_date,
    orders.user_id AS orders_user_id,
    users.user_id AS users_user_id,
```

```
    users.name
FROM users
INNER JOIN orders ON orders.user_id = users.user_id
ORDER BY order_id;
```

- 이번에는 조인의 위치를 변경해서 `FROM`에 `users`를 `JOIN`에 `orders`를 사용했다.

[실행 결과]

order_id	order_date	orders_user_id	users_user_id	name
1	2025-06-10 10:00:00	1	1	션
2	2025-06-10 10:05:00	1	1	션
3	2025-06-11 14:20:00	2	2	네이트
4	2025-06-12 09:00:00	3	3	세종대왕
5	2025-06-15 11:30:00	4	4	이순신
6	2025-06-16 18:00:00	5	5	마리 퀴리
7	2025-06-17 12:00:00	2	2	네이트

- 결과는 위치를 변경하기 전의 쿼리와 **완전히 동일**하다.

내부 조인에서 데이터베이스는 `ON` 절의 `users.user_id = orders.user_id` 조건을 만족하는 짝을 찾을 뿐, 어느 테이블을 먼저 읽었는지는 최종 결과에 영향을 주지 않는다.

내부 조인 3

실무 팁: 조인 순서는 언제 중요할까?

내부 조인에서는 결과가 같으므로 어떤 순서로 작성해도 무방하다. 하지만 쿼리를 읽는 사람의 입장에서 **어떤 데이터가 중심이 되는가**에 따라 순서를 정하면 가독성이 높아진다.

- **주문 목록**을 중심으로 고객 정보를 추가하고 싶다면 `FROM orders JOIN users`
- **고객 목록**을 중심으로 주문 정보를 조회하고 싶다면 `FROM users JOIN orders`

이렇게 같이 작성하는 것이 논리의 흐름을 이해하기 더 쉽다. 지금 우리의 문제는 "주문 완료 건에 대한 고객 정보"를 찾는 것이므로 `FROM orders`로 시작하는 것이 조금 더 자연스럽다.

> **나중에 배울 `OUTER JOIN`에서는 조인 순서가 결과에 매우 큰 영향을 미친다. 하지만 `INNER JOIN`에서는 교집합을 선택하기 때문에 순서와 상관없이 결과는 항상 같다.**

조인은 우리가 원하는 데이터가 여러 테이블에 흩어져 있을 때, 그 정보들을 하나로 모으는 가장 기본적인 방법이다.

그런데 여기서 또 다른 질문이 생긴다.
"그렇다면, 우리 쇼핑몰에 가입은 했지만 아직 한 번도 주문하지 않은 고객은 어떻게 찾아낼 수 있을까?"
내부 조인(`INNER JOIN`)은 양쪽에 모두 데이터가 있는 경우만 보여주기 때문에 이 질문에는 답할 수 없다.
이 문제를 해결하기 위해 다음 시간에는 외부 조인(`OUTER JOIN`)에 대해 알아보겠다.

그 전에 잠깐 실무에서 사용하는 테이블 별칭 팁을 알아보자.

가독성을 높이는 테이블 별칭(Alias)

앞선 쿼리들을 보면 `users.user_id`, `orders.status` 처럼 테이블 이름을 계속 반복해서 작성해야 했다. 쿼리가 길어지고 복잡해질수록 이는 매우 번거롭고 가독성을 떨어뜨린다.

이럴 때 사용하는 것이 바로 **테이블 별칭(Alias)**이다. `AS` 키워드를 사용하거나, `AS`를 생략하고 한 칸 띄우고 원하는 별칭을 붙여주면 된다.

```
SELECT
    u.user_id,
    u.name,
    o.order_date
FROM orders AS o
INNER JOIN users AS u ON o.user_id = u.user_id
WHERE o.status = 'COMPLETED';
```

- `orders AS o`: `orders` 테이블에 `o`라는 별칭을 붙였다.
- `users AS u`: `users` 테이블에 `u`라는 별칭을 붙였다.
- 이제 `orders.status` 대신 `o.status`, `users.name` 대신 `u.name`과 같이 간결하게 컬럼을 지정할 수 있다.

실무에서는 `AS` 생략

실무에서는 테이블 별칭을 붙일 때 `AS` 키워드를 생략하는 경우가 더 많다. 테이블 별칭의 `AS`를 생략하고, 추가로 `INNER JOIN`의 `INNER`도 생략했다.

```
SELECT
    u.user_id,
    u.name,
    o.order_date
FROM orders o
JOIN users u ON o.user_id = u.user_id
WHERE o.status = 'COMPLETED';
```

- `FROM orders o`처럼 테이블 이름 뒤에 한 칸 띄고 별칭을 적는 것만으로도 충분하다. 이 방식이 더 간결하기 때문에 실무 개발자들이 선호하는 스타일이다. 앞으로의 모든 예제에서는 이 방식을 사용할 것이다.
- `INNER JOIN`에서 `INNER`는 생략 가능하므로 `JOIN`만 사용해도 된다. (그냥 `JOIN`이라고 하면 자동으로 `INNER JOIN`이 된다.) 실무에서는 대부분 `INNER JOIN`을 사용할 때 `INNER`는 생략하고 `JOIN`만 사용한다.

[실행 결과]

user_id	name	order_date
1	션	2025-06-10 10:00:00
1	션	2025-06-10 10:05:00
3	세종대왕	2025-06-12 09:00:00
5	마리 퀴리	2025-06-16 18:00:00

별칭을 사용해도 결과는 당연히 동일하다.

실무 팁 - 테이블과 컬럼의 별칭 AS 생략

테이블 별칭: AS 생략

테이블 이름 뒤에 오는 별칭은 문법적으로 혼동의 여지가 거의 없다. `FROM table_name alias_name`과 같이 `AS` 없이 써도 코드를 읽고 해석하는 데 문제가 되지 않는다. 따라서 코드를 더 간결하게 만들기 위해 `AS`를 생략하는 경우가 많다.

예를 들어, `FROM employees e`는 `FROM employees AS e`와 동일하게 작동하며, 더 짧고 깔끔하게 보인다.

컬럼 별칭: AS 사용

반면, 컬럼 목록에서는 `AS`를 생략하면 어떤 것이 원래 컬럼 이름이고 어떤 것이 별칭인지 즉시 파악하기 어려울 수 있다. 특히 여러 컬럼을 나열하거나 복잡한 함수를 사용할 때 `AS`를 명시적으로 써주면 코드의 의도를 명확하게 전달하여 가독성을 크게 향상시킨다.

가독성이 낮은 예시

```
SELECT salary * 12 annual_salary
```

가독성이 높은 예시

```
SELECT salary * 12 AS annual_salary
```

이처럼 **AS**를 사용하면 `salary * 12`의 결과를 `annual_salary`라는 이름으로 부른다는 의미가 훨씬 명확해진다.

이러한 관행은 필수는 아니지만 많은 개발자가 코드의 가독성과 유지보수성을 높이기 위해 따르는 일종의 **코딩 컨벤션(약속)**이다.

> ✨ **실무 팁 정리**
>
> - 테이블에 별칭으로 사용하는 AS는 주로 생략한다.
> - 컬럼에 별칭으로 사용하는 AS는 가독성을 위해 생략하지 않고 사용한다.

문제와 풀이

> 💬 **문제와 풀이 진행 방법**
>
> 1. **반드시 스스로 문제를 풀어보자**
> 2. 문제를 10분 이상 고민한다면 기존 내용을 복습하고 문제를 풀어보자.
> 3. 그래도 문제가 풀리지 않으면 정답을 보고 문제를 푼 이후에, 스스로 문제를 다시 풀어보자.

문제1: 주문별 상품 정보 조회

[문제]

`INNER JOIN`을 사용하여 `orders` 테이블과 `products` 테이블을 연결해라. 모든 주문에 대해 `주문 ID`, `상품명`, `주문 수량`이 포함된 목록을 조회하는 SQL을 작성하고 `order_id` 오름차순 정렬해라. 가독성을 위해 **테이블 별칭**을 사용해야 한다.

[실행 결과]

order_id	name	quantity
1	프리미엄 게이밍 마우스	1
2	관계형 데이터베이스 입문	2
3	기계식 키보드	1
4	관계형 데이터베이스 입문	1
5	4K UHD 모니터	1
6	프리미엄 게이밍 마우스	1
7	프리미엄 게이밍 마우스	2

[정답]

```sql
SELECT
    o.order_id,
    p.name,
    o.quantity
FROM
    orders o
JOIN
    products p ON o.product_id = p.product_id
ORDER BY
    o.order_id;
```

> **영상 수정**
> 영상에서는 SELECT o.product_id로 입력했는데, 메뉴얼의 정답과 같이 o.order_id를 사용해야 합니다.

문제2: 3개 테이블 조인하기

[문제]

`orders`, `users`, `products` 세 개의 테이블을 모두 조인해라. `SHIPPED` (배송) 상태인 주문에 대해 `주문 ID`, `고객 이름`, `상품명`, `주문 날짜`를 조회하는 SQL을 작성해라.

[실행 결과]

order_id	user_name	product_name	order_date
3	네이트	기계식 키보드	2025-06-11 14:20:00
7	네이트	프리미엄 게이밍 마우스	2025-06-17 12:00:00

[정답]

```
SELECT
    o.order_id,
    u.name AS user_name,
    p.name AS product_name,
    o.order_date
FROM
    orders o
JOIN
    users u ON o.user_id = u.user_id
JOIN
    products p ON o.product_id = p.product_id
WHERE
    o.status = 'SHIPPED';
```

문제3: 고객별 총 구매액 계산

[문제]

`INNER JOIN`과 집계 함수를 함께 사용해서, 각 고객이 지금까지 주문한 총 구매액을 계산해라. 결과는 `고객 이름(user_name)`과 `총 구매액(total_purchase_amount)`으로 구성되어야 하며, 총 구매액이 높은 순서대로 정렬해야 한다. (`총 구매액 = 주문수량 * 상품가격`)

[실행 결과]

user_name	total_purchase_amount
이순신	350000
네이트	270000
션	131000
마리 퀴리	75000
세종대왕	28000

[정답]

```sql
SELECT
    u.name AS user_name,
    SUM(o.quantity * p.price) AS total_purchase_amount
FROM
    orders o
JOIN
    users u ON o.user_id = u.user_id
JOIN
    products p ON o.product_id = p.product_id
GROUP BY
    u.name
ORDER BY
    total_purchase_amount DESC;
```

정리

조인이 필요한 이유

- 데이터는 정규화 과정을 거쳐 여러 테이블에 나뉘어 저장된다.
- 데이터 중복을 피하고 데이터 갱신 삽입 삭제 시 발생할 수 있는 이상 현상을 방지하기 위함이다.
- 분리된 테이블에서 의미 있는 정보를 얻으려면 흩어진 데이터를 다시 연결해야 한다.
- 조인은 여러 테이블에 흩어진 데이터를 기본 키와 외래 키를 기준으로 합쳐서 보여주는 핵심 기술이다.

내부 조인 1

- 내부 조인(`INNER JOIN`)은 두 테이블에 공통으로 존재하는 데이터만 결과로 보여준다.
- `ON` 절에 명시된 연결 조건이 참이 되는 행들만 결과에 포함시킨다.
- 조인의 논리적 처리 순서는 `FROM/JOIN`으로 테이블을 결합하고 `WHERE`로 조건을 필터링한 후 `SELECT`로 원하는 컬럼을 선택하는 순서로 진행된다.
- 내부 조인에서 `INNER` 키워드는 생략하고 `JOIN`만 사용해도 된다.

내부 조인 2

- 내부 조인은 집합의 관점에서 두 테이블의 교집합을 찾는 것과 같다.
- 두 테이블 양쪽에 모두 연결고리가 있는 데이터만 결과에 포함된다.
- 한쪽 테이블에만 존재하는 데이터 예를 들어 주문 기록이 없는 고객은 조인 결과에서 제외된다.
- 내부 조인은 조인하는 테이블의 순서를 바꿔도 항상 같은 결과를 반환한다.

내부 조인 3

- 테이블 별칭 `Alias`을 사용하면 쿼리의 가독성을 높이고 간결하게 작성할 수 있다.
- 실무에서는 보통 테이블 별칭의 `AS`는 생략하고 컬럼 별칭의 `AS`는 명시적으로 사용한다.
- 내부 조인은 순서에 상관없이 결과가 같지만 나중에 배울 외부 조인은 순서가 결과에 큰 영향을 미친다.