3. 조인2 - 외부 조인과 기타 조인

#0.강의/2.데이터베이스로드맵/2.기본

- /외부 조인 1
- /외부 조인 2
- /조인의 특징
- /셀프 조인
- /CROSS 조인
- /조인 종합 실습
- /문제와 풀이1
- /문제와 풀이2
- /정리

외부 조인 1

내부 조인(`INNER JOIN`)을 통해 우리는 양쪽 테이블에 모두 존재하는, 짝이 맞는 데이터들을 성공적으로 연결했다. 하지만 실무에서는 종종 짝이 없는, 소외된 데이터를 찾아야 할 때가 있다.

지난 시간에 던졌던 질문을 다시 떠올려 보자.
"우리 쇼핑몰에 가입은 했지만, 아직 한 번도 주문하지 않은 고객은 누구일까?"
또는 이런 질문도 있을 수 있다.
"야심차게 출시했지만, 아직 단 한 번도 팔리지 않은 비운의 상품은 무엇일까?"

이 질문들에 내부 조인은 답할 수 없다. 왜냐하면 내부 조인은 주문 기록이 있는 고객, 판매 기록이 있는 상품, 즉 `orders` 테이블과 짝이 맞는 데이터만 보여주기 때문이다. 주문한 적 없는 고객이나 팔린 적 없는 상품은 `orders` 테이블에 짝이 없으므로 결과에서 아예 누락된다. 이런 경우 왜 내부 조인을 사용할 수 없는지 예제로 확인해보자.

실습: '한 번도 주문하지 않은 고객' 찾기

'한 번도 주문하지 않은 고객'을 찾는 문제를 단계별로 접근해보자. 먼저 각 테이블을 개별적으로 조회해서 어떤 데이터가 있는지 확인하고, 왜 내부 조인으로는 해결할 수 없는지 이해해보자.

1단계: 각 테이블 개별 조회하기

먼저 `users` 테이블과 `orders` 테이블을 각각 조회해서 어떤 고객이 있고, 어떤 주문이 있는지 직접 확인해보자.

```sql
-- users 테이블 조회 (최소한의 필드만)
SELECT user_id, name, email FROM users;
```

[실행 결과 - users]

user_id	name	email
1	션	sean@example.com
2	네이트	nate@example.com
3	세종대왕	sejong@example.com
4	이순신	sunsin@example.com
5	마리 퀴리	marie@example.com
6	레오나르도 다빈치	vinci@example.com

```sql
-- orders 테이블 조회 (최소한의 필드만)
SELECT user_id, order_id FROM orders
order by user_id;
```

[실행 결과 - orders]

user_id	order_id
1	1
1	2
2	3
2	7
3	4
4	5

5	6

이제 눈으로 직접 비교해보자. `users` 테이블에는 `user_id`가 `{1, 2, 3, 4, 5, 6}`인 고객이 있다. 그런데 `orders` 테이블에는 `user_id`가 `{1, 2, 3, 4, 5}`인 주문만 있고, `user_id`가 6인 주문은 없다. 즉, '레오나르도 다빈치'는 가입은 했지만 한 번도 주문하지 않은 고객이다.

이렇게 눈으로 각각의 테이블을 하나하나 비교하는 것은 너무 많은 시간이 걸린다. 조인을 사용해보자.

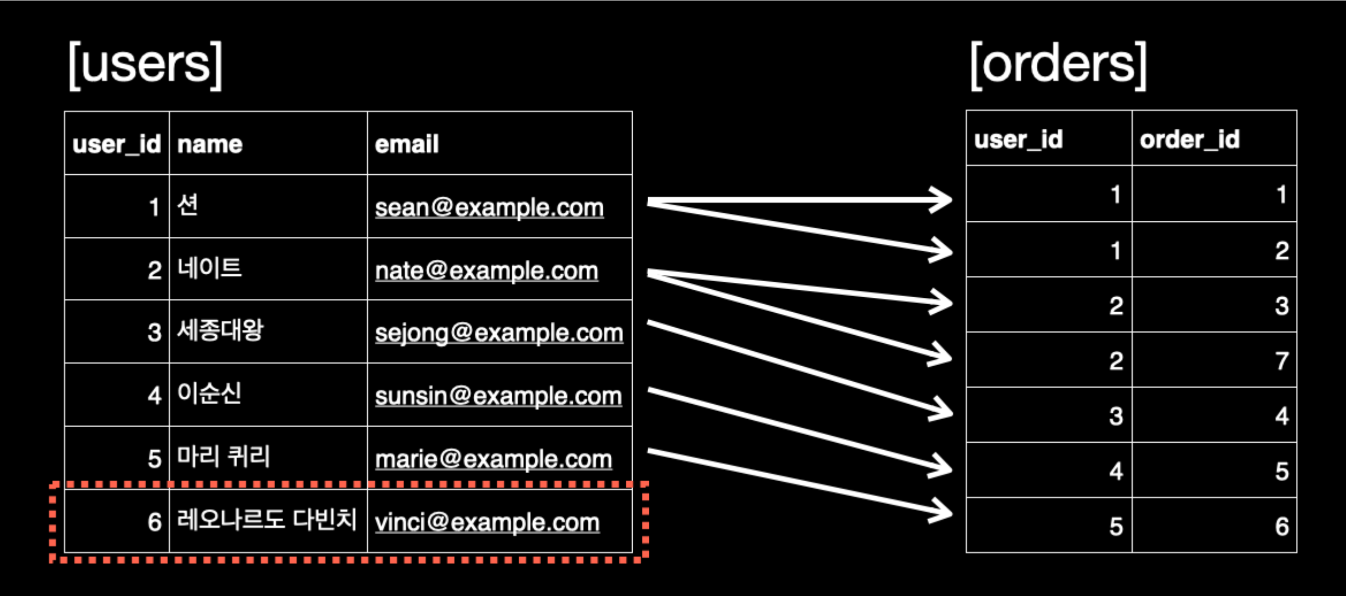
2단계: 내부 조인으로 시도해보기

결과를 한 번에 쉽게 확인하기 위해 내부 조인(`INNER JOIN`)을 사용해서 고객과 주문을 연결해보자.

```
SELECT
    u.user_id,
    u.name,
    o.user_id,
    o.order_id
FROM users u
JOIN orders o ON u.user_id = o.user_id
ORDER BY u.user_id;
```

- `INNER`는 생략했다. `INNER`를 생략하고 `JOIN`만 적으면 내부 조인(`INNER JOIN`)으로 작동한다.

[조인 과정]



- 조인 과정을 보면 레오나르도 다빈치의 `users.user_id:6`에 해당하는 `orders.user_id:6`이 없다.
- 따라서 레오나르도 다빈치는 조인 대상에서 제외된다.

[실행 결과]

user_id	name	user_id	order_id
1	션	1	1
1	션	1	2
2	네이트	2	3
2	네이트	2	7
3	세종대왕	3	4
4	이순신	4	5
5	마리 퀴리	5	6

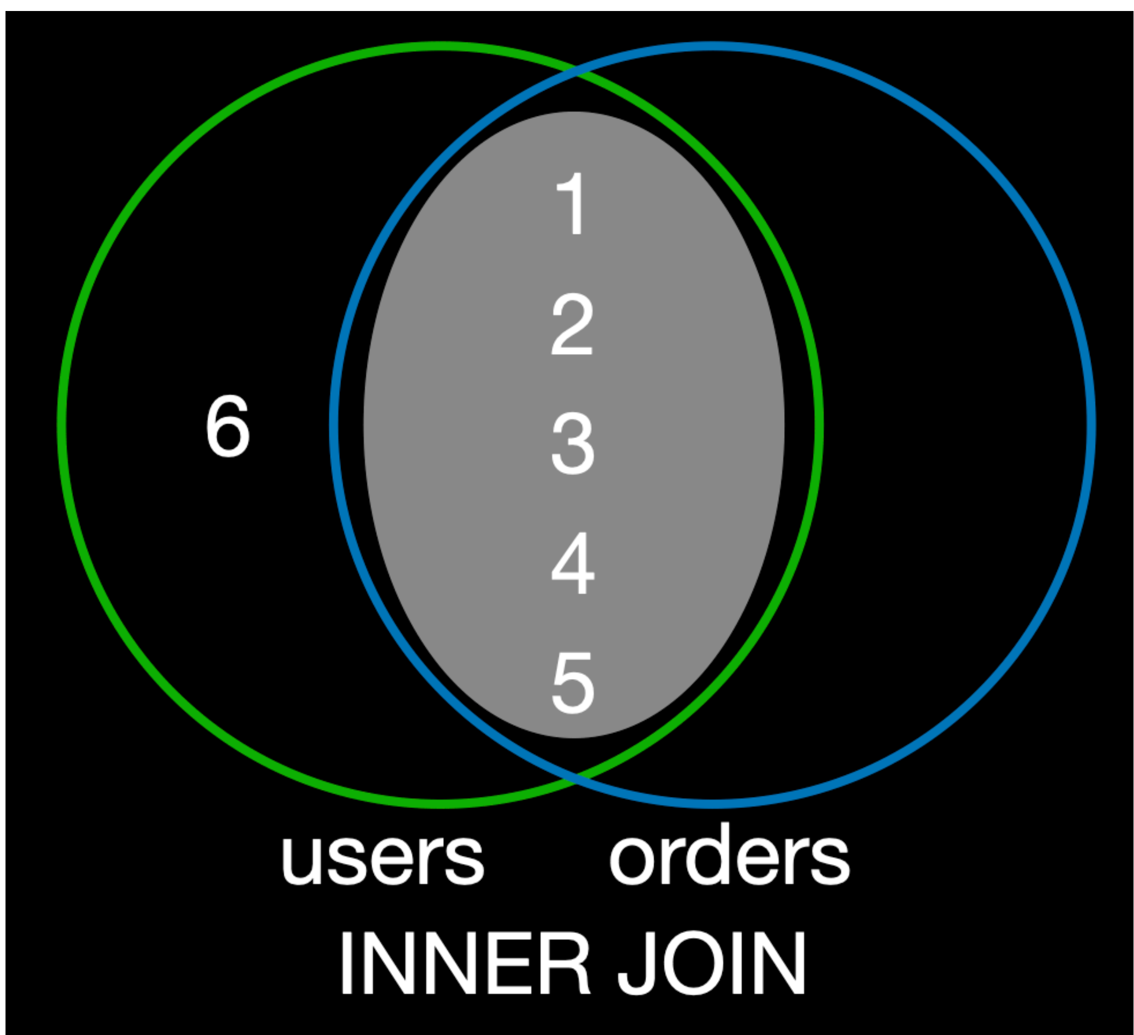
결과를 보면 '레오나르도 다빈치'(`user_id` : 6)가 결과에서 완전히 사라졌다. 내부 조인(`INNER JOIN`)은 양쪽 테이블에 모두 존재하는 데이터만 보여주기 때문이다. 즉 주문 기록이 없는 고객은 `orders` 테이블에 짝이 없어서 결과에서 제외된다.

외부 조인(OUTER JOIN)의 필요성
내부 조인(`INNER JOIN`)으로는 주문 기록이 없는 고객을 찾을 수 없다. 이런 경우에는 외부 조인(`OUTER JOIN`)이 필요하다. 외부 조인을 사용하면 한쪽 테이블에만 존재하는 데이터도 결과에 포함시킬 수 있다.

이처럼 한쪽에는 데이터가 있지만, 다른 한쪽에는 없는 데이터까지 모두 포함해서 보고 싶을 때 사용하는 기술이 바로 외부 조인이다.

외부 조인의 필요성 및 개념

조인은 본질적으로 두 테이블의 집합에 대한 이야기다.
그 중에 내부 조인은 교집합이다.

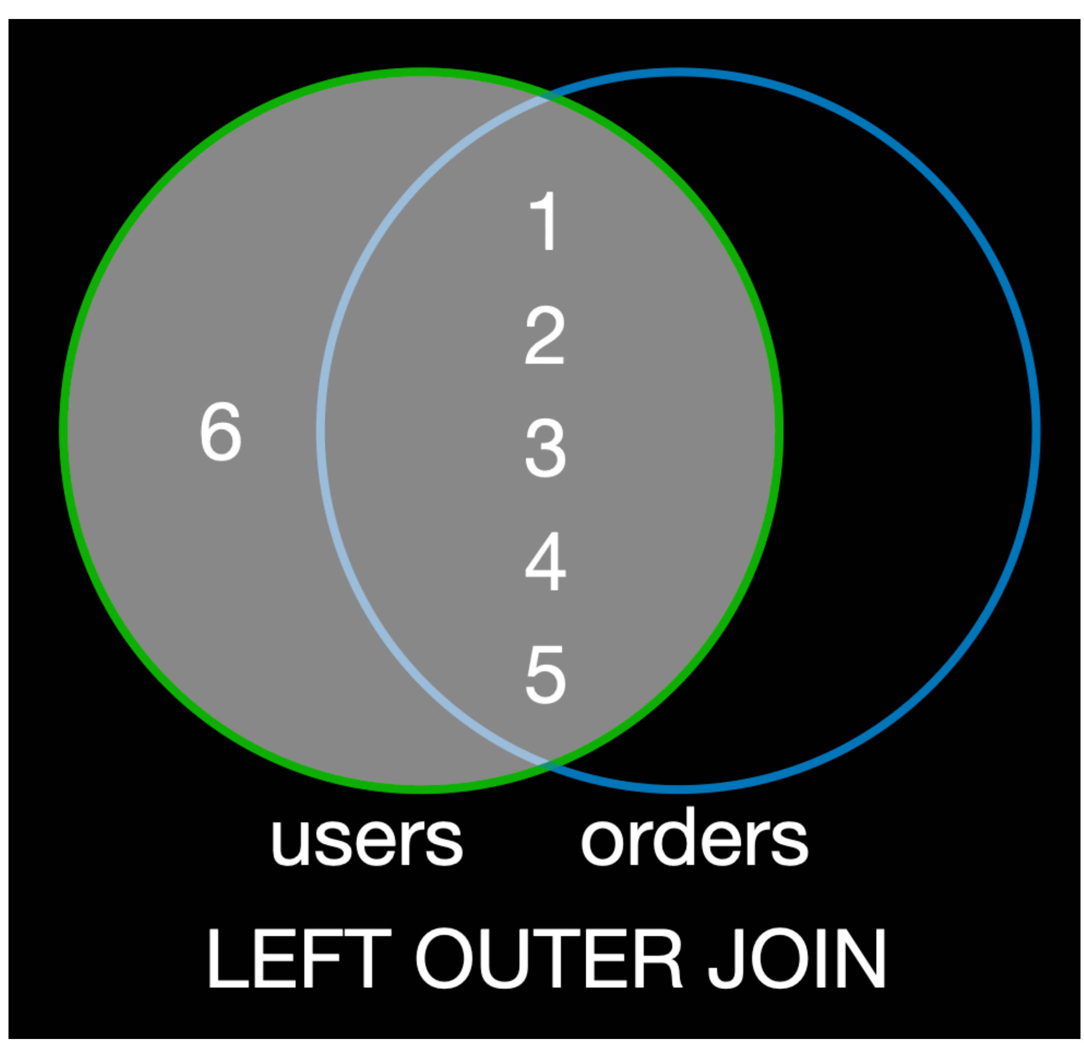


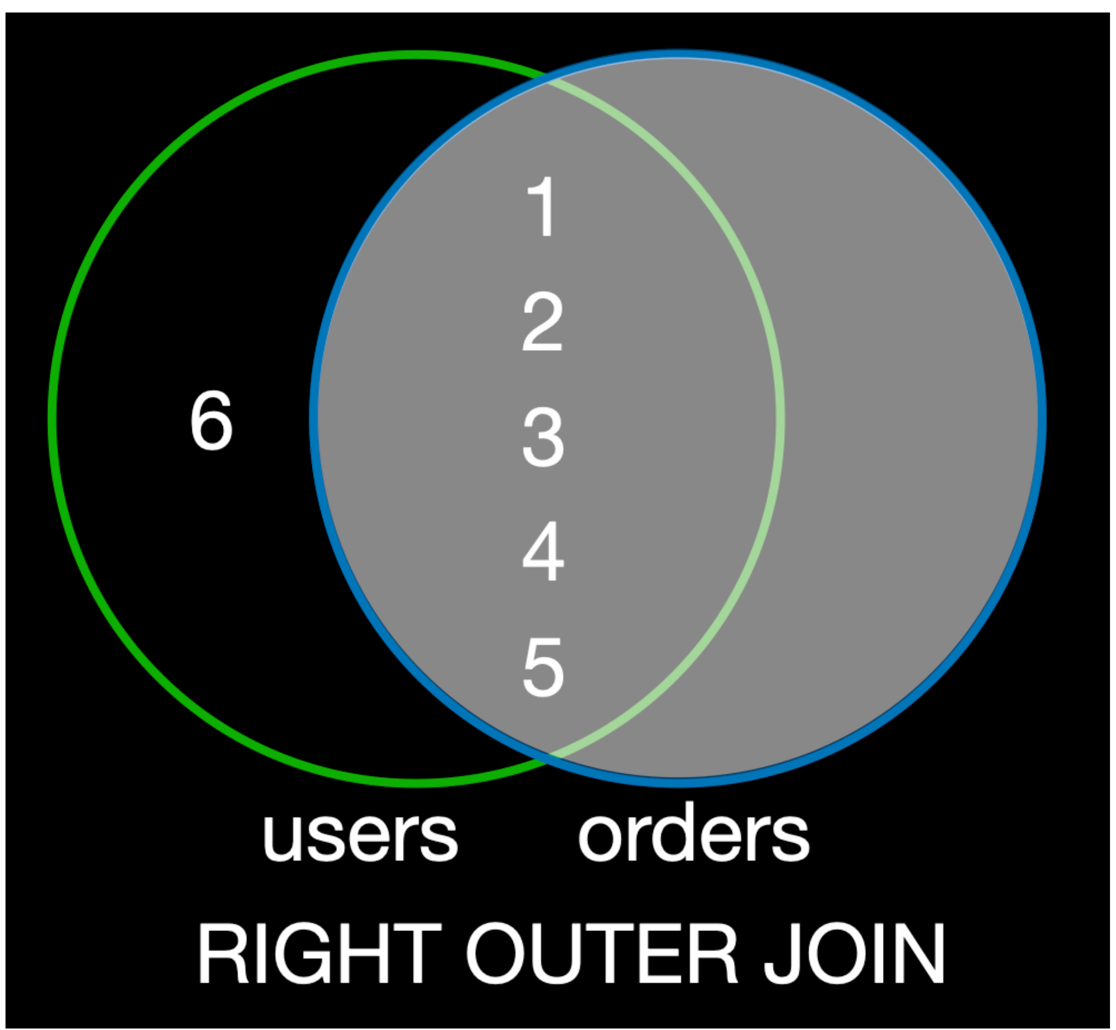
- 내부 조인은 교집합 영역이기 때문에 `users`에만 속한 '레오나르도 다빈치'(`user_id`: 6)를 결과에 포함할 수 없다.

그렇다면 **교집합 영역 뿐만 아니라 왼쪽 영역을 모두 포함하는 집합 또는 오른쪽 영역을 모두 포함하는 집합 또는 양쪽을 모두 포함하는 집합은 어떻게 선택할 수 있을까?**
예를 들어서 레오나르도 다빈치'(`user_id`: 6)를 결과에 포함하려면 어떻게 해야할까?

외부 조인의 필요성

외부 조인(`OUTER JOIN`)은 두 테이블을 조인할 때, 특정 테이블의 데이터는 `ON` 조건이 맞지 않더라도 모두 결과에 포함시키는 방법이다.


1
2
3
4
5
6
users
orders
LEFT OUTER JOIN

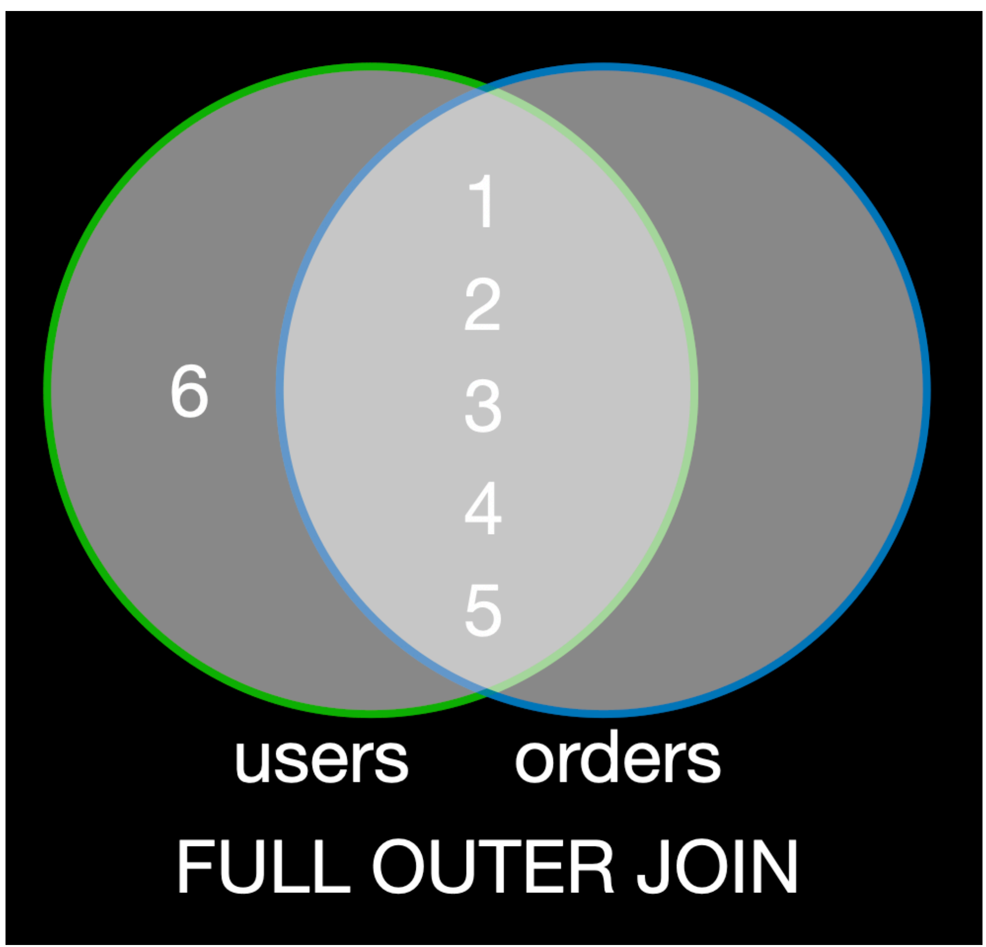


이때 기준이 되는 테이블이 어느 쪽이냐에 따라 `LEFT OUTER JOIN`과 `RIGHT OUTER JOIN`으로 나뉜다.

그리고 **교집합 영역**은 물론이고, **기준이 되는 테이블의 데이터는 결과에 모두 포함**된다.

레오나르도 다빈치'(`user_id`: 6)를 결과에 포함하려면 `LEFT OUTER JOIN`을 선택하면 된다.

> **용어 - 외부 조인 (OUTER JOIN)**
>
> 내부 조인(INNER JOIN)은 교집합 안(INNER)을 의미한다.
>
> 외부 조인은 교집합 밖(OUTER)의 영역을 포함한다는 의미다.



> **풀 외부 조인(FULL OUTER JOIN)**
> 양쪽 모두를 함께 포함하는 풀 외부 조인이라는 방법도 있다.
> 실무에서 잘 사용하지 않고, MySQL은 지원하지 않는다.

`LEFT OUTER JOIN` vs `RIGHT OUTER JOIN`

외부 조인의 핵심은 '기준 테이블'을 정하는 것이다. `LEFT OUTER JOIN` 은 왼쪽 테이블을, `RIGHT OUTER JOIN` 은 오른쪽 테이블을 기준으로 삼는다.

> **OUTER는 생략 가능**
> 다음과 같이 OUTER는 생략해서 사용할 수 있다. 실무에서는 생략하는 것을 권장한다.
> - `LEFT OUTER JOIN` → `LEFT JOIN`
> - `RIGHT OUTER JOIN` → `RIGHT JOIN`

- `LEFT JOIN` **(또는** `LEFT OUTER JOIN` **)**
  - `LEFT JOIN` 구문의 **왼쪽(FROM 절)에 있는 테이블**이 기준이 된다.

- 일단 왼쪽 테이블의 모든 데이터를 결과에 포함시킨다.
- 그다음, `ON` 조건에 맞는 데이터를 오른쪽 테이블에서 찾아 옆에 붙여준다.
- 만약 오른쪽 테이블에 짝이 맞는 데이터가 없다면, 그 자리는 `NULL` 값으로 채워진다.

- `RIGHT JOIN` **(또는** `RIGHT OUTER JOIN`**)**
  - `RIGHT JOIN` 구문의 **오른쪽(JOIN 절)에 있는 테이블**이 기준이 된다.
  - 일단 오른쪽 테이블의 모든 데이터를 결과에 포함시킨다.
  - 그다음, `ON` 조건에 맞는 데이터를 왼쪽 테이블에서 찾아 붙인다.
  - 마찬가지로 왼쪽 테이블에 짝이 맞는 데이터가 없다면, 그 자리는 `NULL` 로 채워진다.

실습 1: `LEFT JOIN`으로 '한 번도 주문하지 않은 고객' 찾기

첫 번째 질문, "한 번도 주문하지 않은 고객은 누구인가?"에 답을 찾아보자. 이 질문의 기준은 '고객'이다. 따라서 `users` 테이블이 `LEFT JOIN`의 왼쪽에 위치해야 한다.

1단계: `users`**를 기준으로** `orders` **테이블** `LEFT JOIN` **하기**

먼저 `users` 테이블의 모든 고객을 일단 다 보여주고, 오른쪽에 주문 정보를 붙여보자. 어떤 결과가 나오는지 직접 확인하는 것이 중요하다.

```sql
SELECT
    u.user_id,
    u.name,
    o.user_id,
    o.order_id
FROM users u
LEFT JOIN orders o ON u.user_id = o.user_id
ORDER BY u.user_id;
```

- `LEFT JOIN`을 사용했다.
- `users`가 왼쪽이고, `orders`가 오른쪽이다.
  - SQL을 보면 FROM 다음에 `users`가 먼저 나오고 그 다음에 `orders`가 나온다.

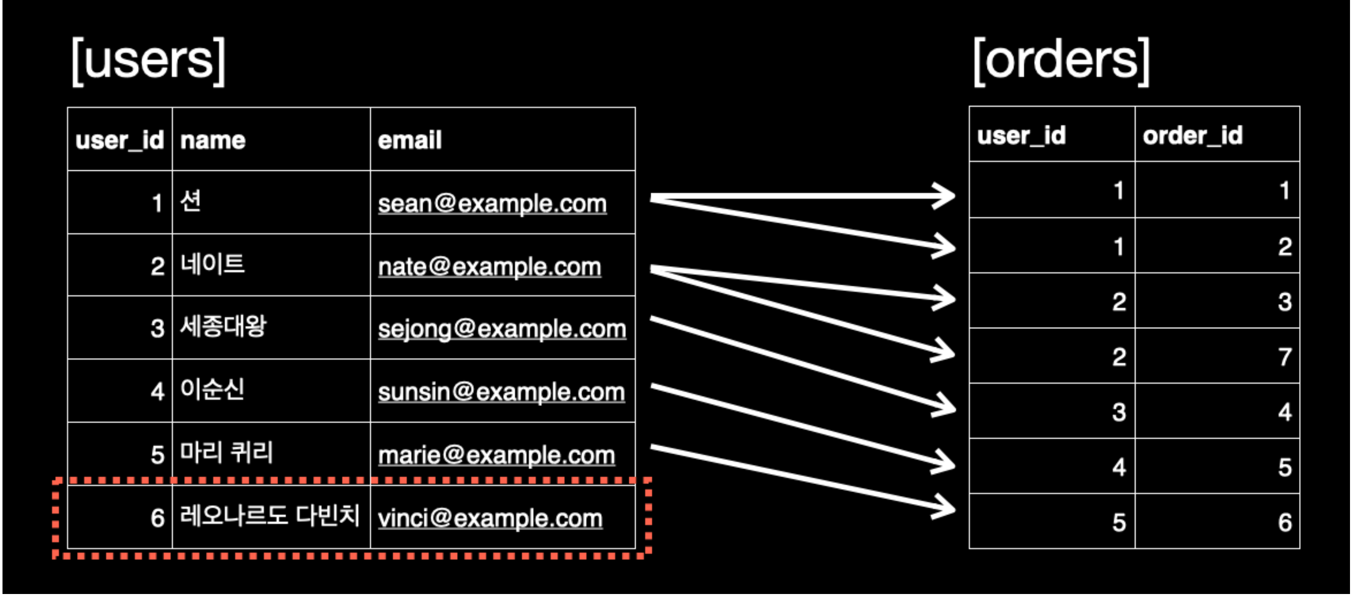
[실행 결과]

user_id	name	user_id	order_id
1	션	1	1

1	션	1	2
2	네이트	2	3
2	네이트	2	7
3	세종대왕	3	4
4	이순신	4	5
5	마리 퀴리	5	6
6	레오나르도 다빈치	NULL	NULL

실행 결과를 보자. `INNER JOIN` 에서는 보이지 않던 '레오나르도 다빈치(`user_id:6`)'가 드디어 나타났다. 그는 주문 기록이 없기 때문에, `orders` 테이블에서 가져온 `user_id` 와 `order_id` 컬럼의 값이 모두 `NULL` 로 채워져 있다. 이것이 바로 `OUTER JOIN` 이다.

[LEFT OUTER 조인 과정]



- `LEFT JOIN` 은 왼쪽에 있는 기준이 되는 테이블(`users`)의 모든 행을 포함한다.
- 참고로 여기서 레오나르도 다빈치(`user_id:6`)는 `orders` 에 조인할 대상이 없다.

[조인 결과 - LEFT JOIN]

[users LEFT JOIN orders]

user_id	name	email	user_id	order_id
1	션	sean@example.com	1	1
1	션	sean@example.com	1	2
2	네이트	nate@example.com	2	3
3	세종대왕	sejong@example.com	2	4
4	이순신	sunsin@example.com	3	5
5	마리 퀴리	marie@example.com	4	6
2	네이트	nate@example.com	5	7
6	레오나르도 다빈치	vinci@example.com	NULL	NULL

users 영역 orders 영역

- 이렇게 조인 대상이 없는 경우에는 나머지 값을 `NULL` 로 채우면서 조인 결과에 포함한다.

> **OUTER JOIN은 기준 테이블의 데이터를 모두 포함한다.**

2단계: `NULL` 인 데이터만 필터링하기

주문 정보가 `NULL` 인 고객이 바로 '한 번도 주문하지 않은 고객'이다. `WHERE` 절을 사용해서 이들만 찾아보자.

```
SELECT
    u.user_id,
    u.name,
    u.email
FROM users AS u
LEFT JOIN orders AS o ON u.user_id = o.user_id
WHERE o.order_id IS NULL;
```

NULL 비교

`NULL` 은 '값이 없음'을 나타내는 특별한 상태이므로, `=` 연산자로 비교할 수 없다. 반드시 `IS NULL` 또는 `IS NOT NULL` 을 사용해야 한다.

[실행 결과]

user_id	name	email
6	레오나르도 다빈치	vinci@example.com

드디어 우리 쇼핑몰에서 한 번도 주문하지 않은 고객을 찾아냈다. 이제 이 고객에게만 특별 할인 쿠폰을 보내는 등의 타겟 마케팅을 할 수 있는 근거 데이터가 마련된 것이다.

외부 조인 2

실습 2: `LEFT JOIN` 으로 '단 한 번도 팔리지 않은 상품' 찾기

이번에는 두 번째 질문, "단 한 번도 팔리지 않은 상품은 무엇인가?"에 답해보자. 이번 분석의 기준은 '상품'이다. `products` 테이블의 데이터는 모두 결과에 나와야 한다. 따라서 `products` 테이블이 기준이 되어야 한다.

[products 테이블 주요 정보]

```
SELECT
    product_id,
    name,
    price
FROM products;
```

product_id	name	price
1	프리미엄 게이밍 마우스	75000
2	기계식 키보드	120000
3	4K UHD 모니터	350000
4	관계형 데이터베이스 입문	28000

5	고급 가죽 지갑	150000
6	스마트 워치	280000

[orders 테이블 주요 정보]

```
SELECT
    product_id,
    order_id
FROM orders
order by product_id;
```

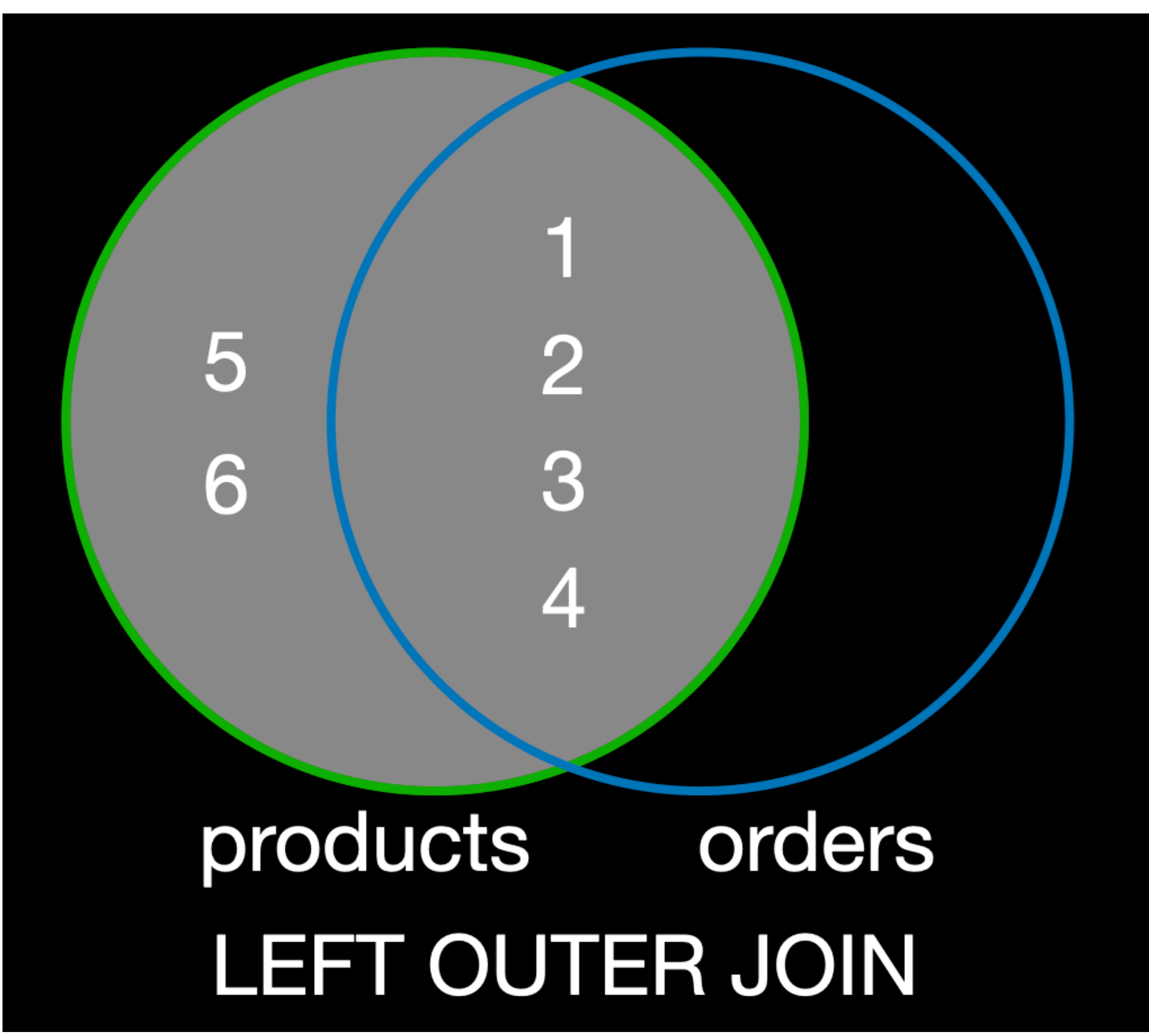
product_id	order_id
1	1
1	6
1	7
2	3
3	5
4	2
4	4

- `product_id` 로 정렬해서 조인을 더 이해하기 쉽게 확인하자.

눈으로 하나하나 비교해보니 `고급 가죽 지갑(product_id:5)`, `스마트 워치(product_id:6)` 상품이 주문 내역에 보이지 않는다.

1단계: `products` 를 기준으로 `orders` 테이블 `LEFT JOIN` 하기

`products` 테이블을 기준으로 `LEFT JOIN` 을 사용해보자.




```
SELECT
    p.product_id,
    p.name,
    p.price,
    o.product_id,
    o.order_id
FROM products p
LEFT JOIN orders o ON p.product_id = o.product_id
```

조인 진행

- 고급 가죽 지갑, 스마트 워치는 주문 내역에 없다.

[실행 결과]

product_id	name	price	product_id	order_id
1	프리미엄 게이밍 마우스	75000	1	1
1	프리미엄 게이밍 마우스	75000	1	6
1	프리미엄 게이밍 마우스	75000	1	7
2	기계식 키보드	120000	2	3
3	4K UHD 모니터	350000	3	5
4	관계형 데이터베이스 입문	28000	4	2
4	관계형 데이터베이스 입문	28000	4	4
5	고급 가죽 지갑	150000	NULL	NULL
6	스마트 워치	280000	NULL	NULL

- `products` 테이블을 기준으로 `LEFT OUTER JOIN` 했으므로 `products`의 제품은 조인 결과에 모두 포함된다.

- 따라서 주문 내역이 없는 고급 가죽 지갑, 스마트 워치도 조인 결과에 포함된다. 이 경우 조인 결과의 주문 관련 필드 값은 `NULL` 이 된다.

2단계: `NULL` 인 데이터만 필터링하기

주문 정보가 `NULL` 인 상품이 바로 '단 한 번도 팔리지 않은 상품'이다. `WHERE` 절을 사용해서 이들만 찾아보자.

```
SELECT
    p.product_id,
    p.name,
    p.price,
    o.product_id,
    o.order_id
FROM products AS p
LEFT JOIN orders AS o ON p.product_id = o.product_id
WHERE o.order_id IS NULL;
```

product_id	name	price	product_id	order_id
5	고급 가죽 지갑	150000	NULL	NULL
6	스마트 워치	280000	NULL	NULL

고급 가죽 지갑, 스마트 워치가 단 한 번도 팔리지 않은 상품인 것을 알 수 있다.

실습 3: `RIGHT JOIN` 으로 '단 한 번도 팔리지 않은 상품' 찾기

이번에는 `RIGHT JOIN` 을 사용해보자.

`RIGHT JOIN` 은 왼쪽이 아니라 오른쪽이 기준이 된다. 따라서 오른쪽 테이블의 데이터가 조인 결과로 모두 나와야 한다.

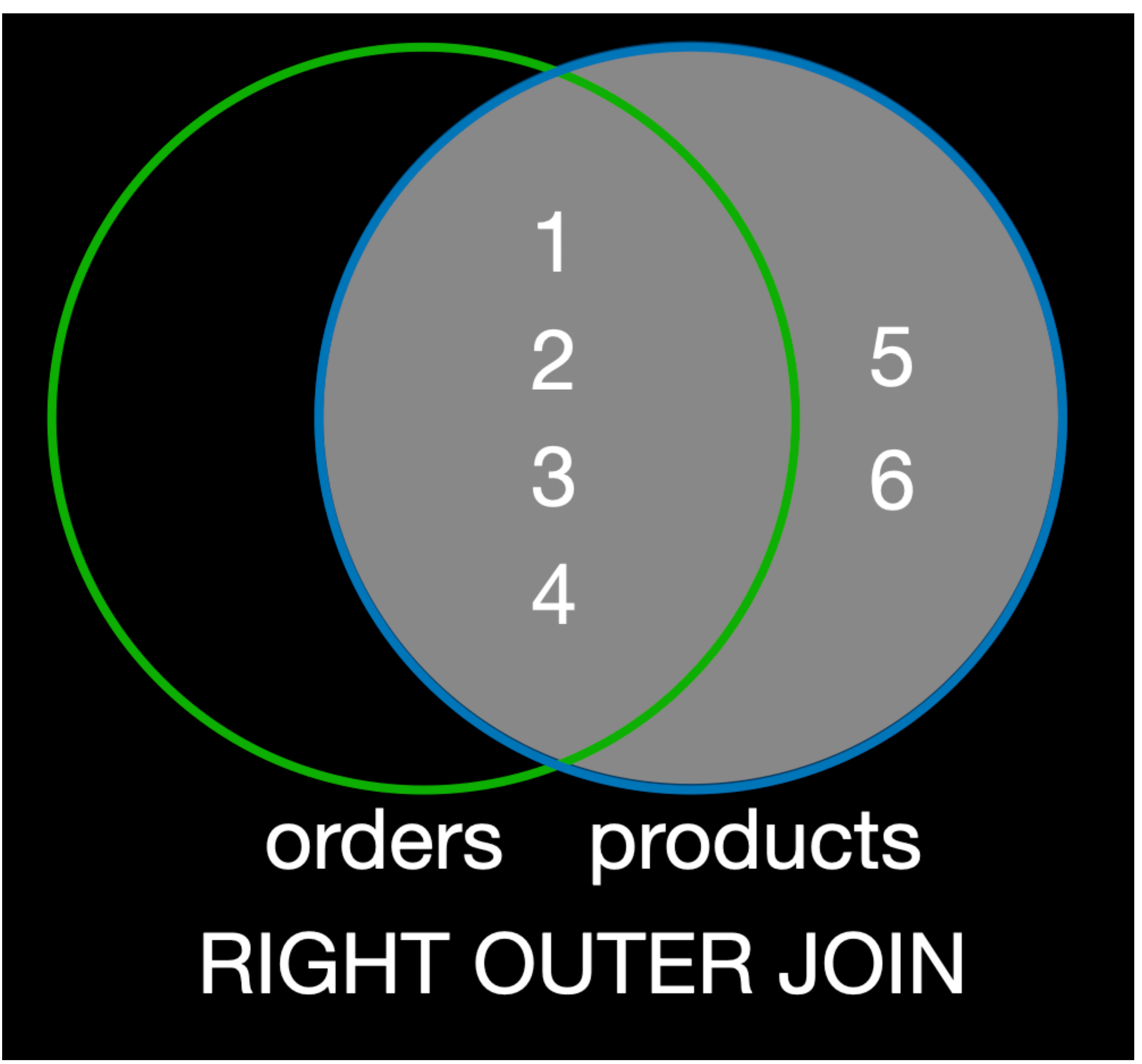
여기서 왼쪽은 FROM 절의 테이블이고, 오른쪽은 JOIN 절의 테이블을 말한다.

`RIGHT JOIN` 을 쉽게 이해하기 위해 앞서 진행했던 예시와 같은 예시를 사용해보자.

앞서 진행했던 예시에서 테이블의 위치만 변경하면 된다. 그리고 이번에도 동일하게 '단 한 번도 팔리지 않은 상품'을 찾으면 된다.

1단계: `products` 를 기준으로 `orders` 테이블 `RIGHT JOIN` 하기

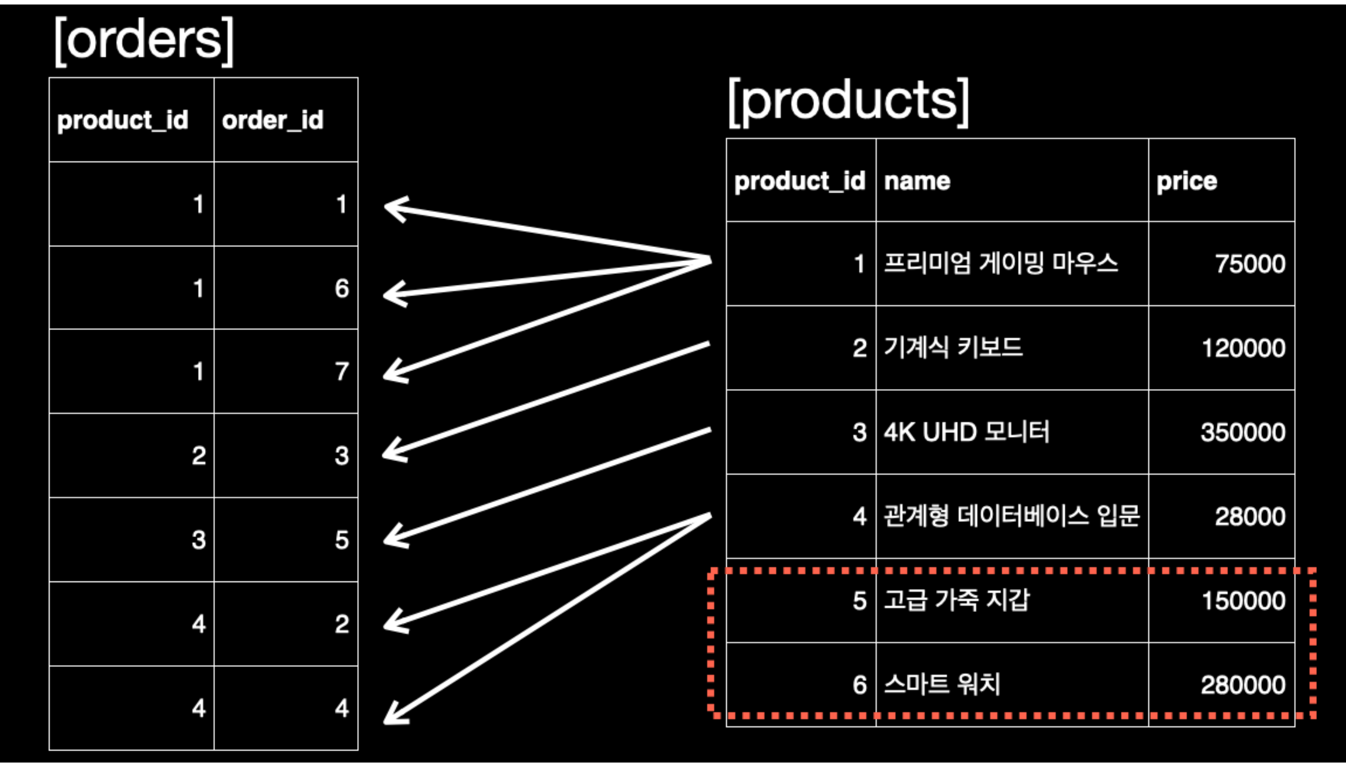
`products` 테이블을 기준으로 `RIGHT JOIN` 을 사용해보자.




```
SELECT
    p.product_id,
    p.name,
    p.price,
    o.product_id,
    o.order_id
FROM orders AS o
RIGHT JOIN products AS p ON o.product_id = p.product_id
```

- 여기서는 `orders` 가 `FROM` 다음에 나오고 `products` 가 `JOIN` 다음에 나온다. 둘의 위치를 바꾸었다.

조인 진행



- 여기서는 `orders` 테이블이 왼쪽, `products` 테이블이 오른쪽이다.
- `RIGHT JOIN`은 오른쪽이 기준이 된다.
- `products.product_id` ➜ `orders.product_id`로 조인한다.

[실행 결과]

product_id	name	price	product_id	order_id
1	프리미엄 게이밍 마우스	75000	1	1
1	프리미엄 게이밍 마우스	75000	1	6
1	프리미엄 게이밍 마우스	75000	1	7
2	기계식 키보드	120000	2	3
3	4K UHD 모니터	350000	3	5
4	관계형 데이터베이스 입문	28000	4	2
4	관계형 데이터베이스 입문	28000	4	4
5	고급 가죽 지갑	150000	NULL	NULL
6	스마트 워치	280000	NULL	NULL

- `RIGHT OUTER JOIN`을 사용했으므로 오른쪽에 있는 `products` 테이블이 기준 테이블이 되고, `products`

테이블은 조인 결과에 모두 포함된다.

- 따라서 주문 내역이 없는 고급 가죽 지갑, 스마트 워치도 조인 결과에 포함된다. 이 경우 조인 결과의 주문 관련 필드 값은 `NULL` 이 된다.

2단계: `NULL` 인 데이터만 필터링하기

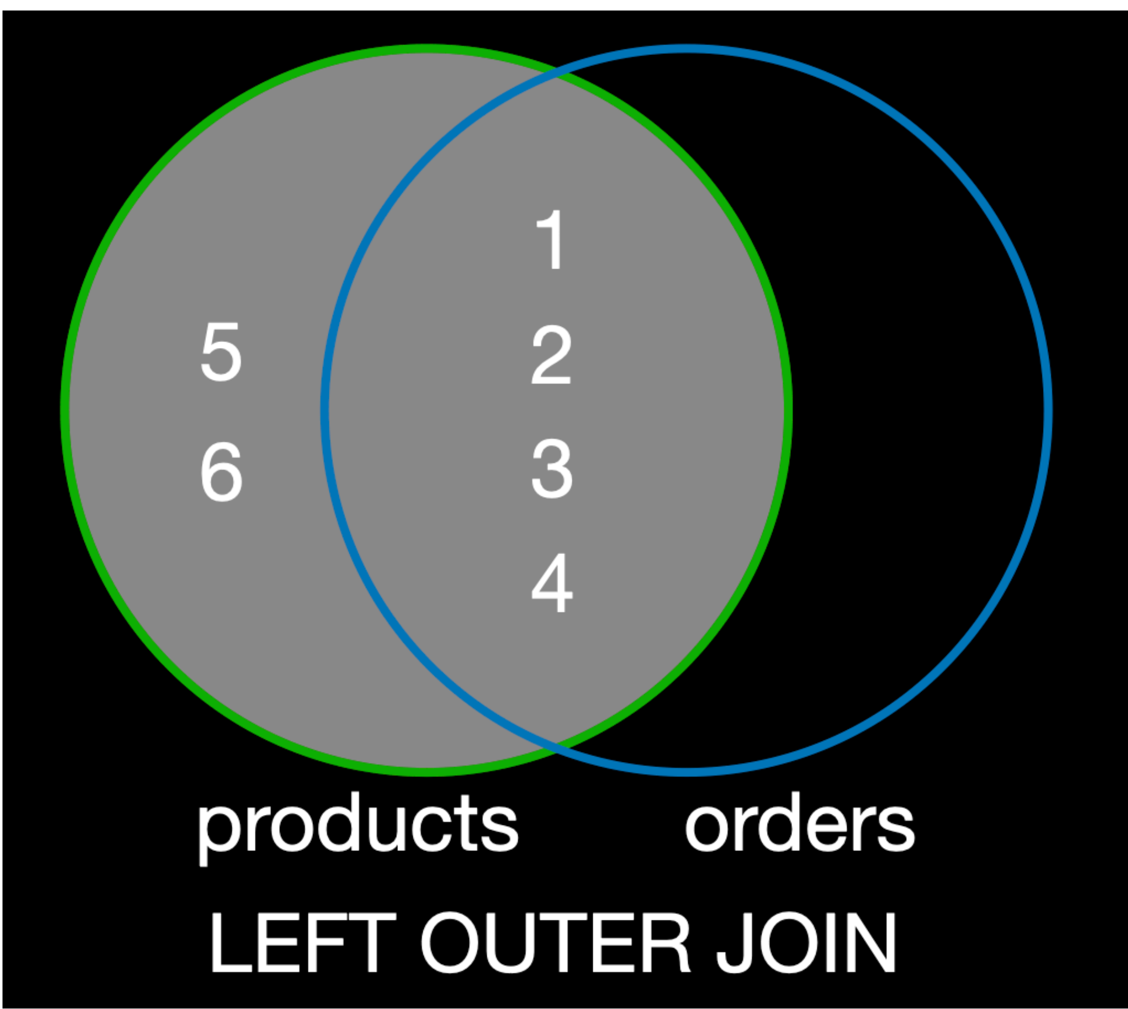
```
SELECT
    p.product_id,
    p.name,
    p.price,
    o.product_id,
    o.order_id
FROM orders AS o
RIGHT JOIN products AS p ON o.product_id = p.product_id
WHERE o.order_id IS NULL;
```

product_id	name	price	product_id	order_id
5	고급 가죽 지갑	150000	NULL	NULL
6	스마트 워치	280000	NULL	NULL

고급 가죽 지갑, 스마트 워치가 단 한 번도 팔리지 않은 상품인 것을 알 수 있다.

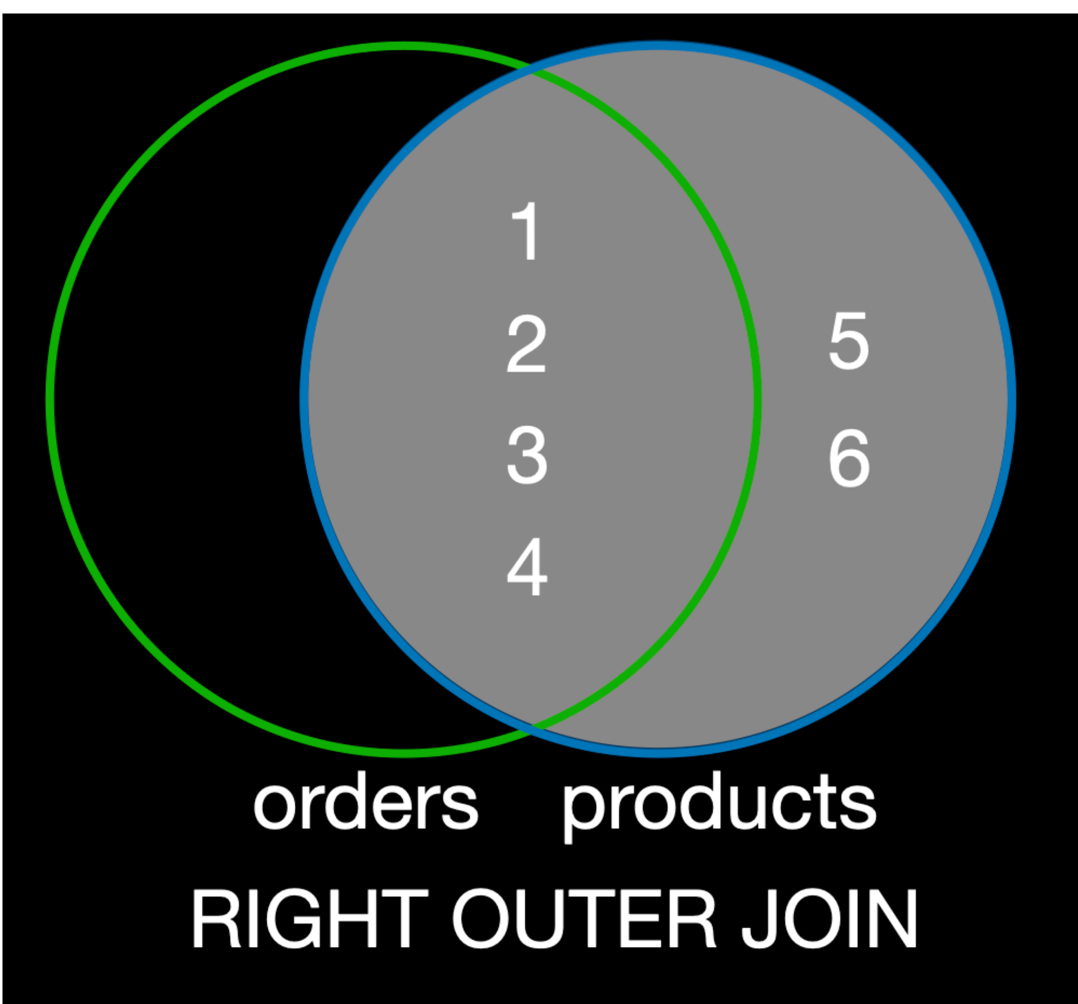
LEFT JOIN과 RIGHT JOIN 선택

앞서 예제를 통해서 알았겠지만, `LEFT JOIN` 과 `RIGHT JOIN` 은 서로 위치를 바꾸어 사용할 수 있다.
다음 다이어그램과 SQL을 비교해보자.




```sql
SELECT
    p.product_id,
    p.name,
    p.price,
    o.product_id,
    o.order_id
FROM products AS p
LEFT JOIN orders AS o ON p.product_id = o.product_id
WHERE o.order_id IS NULL;
```

- `products`는 왼쪽, `orders`는 오른쪽
- `LEFT JOIN`을 사용하면 `products`를 기준 테이블로 정할 수 있다.




```
SELECT
    p.product_id,
    p.name,
    p.price,
    o.product_id,
    o.order_id
FROM orders AS o
RIGHT JOIN products AS p ON o.product_id = p.product_id
WHERE o.order_id IS NULL;
```

- `orders` 는 왼쪽, `products` 는 오른쪽
- `RIGHT JOIN` 을 사용하면 `products` 를 기준 테이블로 정할 수 있다.

두 SQL의 실행 결과는 같다. 테이블을 어떤 위치에 두든 LEFT, RIGHT로 기준 테이블을 정하면 된다.

> ✨ **실무에서는 `LEFT JOIN` 이 `RIGHT JOIN` 보다 훨씬 더 많이 사용된다.**
> 보통 분석의 기준이 되는 테이블을 `FROM` 절에 먼저 쓰고, 필요한 정보를 담은 다른 테이블들을 `LEFT JOIN` 으로 하나씩 붙여나가는 방식으로 쿼리를 작성하는 것이 더 직관적이기 때문이다. `RIGHT JOIN` 은

> 테이블의 순서를 바꾸면 언제나 `LEFT JOIN`으로 동일하게 표현할 수 있다.

정리

- `LEFT JOIN`과 `RIGHT JOIN`은 조인할 때 기준 테이블의 위치를 정한다.
- 사람들은 보통 위에서 아래로, 왼쪽에서 오른쪽으로 글을 읽기 때문에 기준이 되는 내용이 먼저 나오는 것을 선호한다.
- `LEFT JOIN`은 기준 테이블이 `FROM` 절에 먼저 나오므로 더 읽기 편하다.
- 따라서 기준 테이블을 `FROM` 절에 먼저 쓰는 습관을 들이는 것이 좋다.
- `FROM`과 `JOIN` 절에 있는 테이블의 위치를 바꾸면 `LEFT JOIN`을 `RIGHT JOIN`으로 변경할 수 있다.
- 반대로, `FROM`과 `JOIN` 절에 있는 테이블의 위치를 바꾸면 `RIGHT JOIN`을 `LEFT JOIN`으로 변경할 수 있다.
- 이러한 이유로 실무에서는 대부분 `LEFT JOIN`을 사용하며, `RIGHT JOIN`은 잘 사용하지 않는다.

조인의 특징

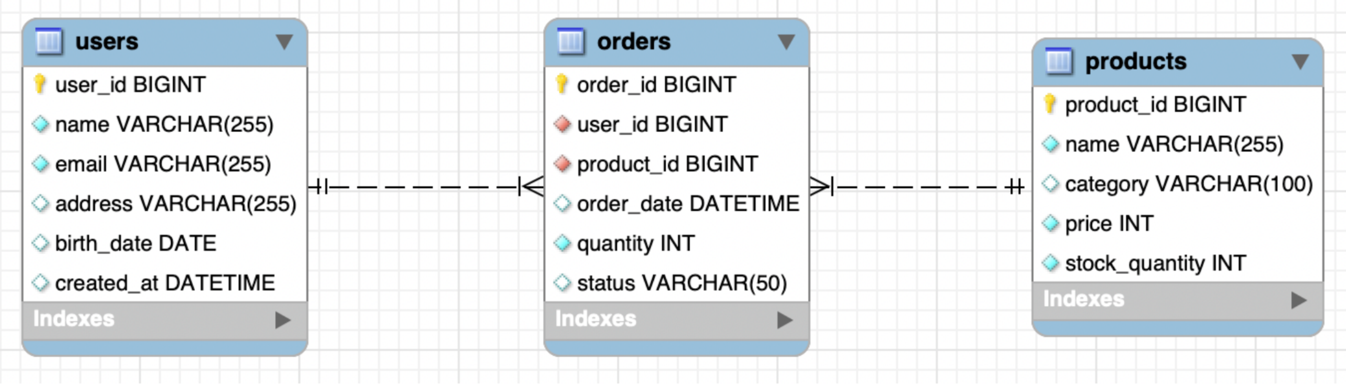
두 테이블을 조인할 때 어떤 경우에는 행이 더 늘어나고 어떤 경우에는 행이 늘어나지 않고 그대로인 경우가 있다. 이 부분은 데이터베이스를 다루는 데 있어 정말 중요하므로 반드시 제대로 이해해야 한다.

조인에서 데이터가 늘어나는 경우

조인은 다음과 같은 특징이 있다.

기준으로 삼는 테이블의 한 행(Row)이 다른 쪽 테이블의 여러 행과 연결될 수 있다면, 결과의 전체 행 수는 늘어난다. 반대로 **한 행이 다른 쪽 테이블의 단 하나의 행과 연결되거나, 아무 행과도 연결되지 않는다면** 행의 수는 늘어나지 않는다.

그렇다면 대체 언제 행이 늘어나고, 언제 그대로일까? 이 원리를 이해하려면 테이블 간의 관계, 특히 **기본 키(Primary Key, PK)**와 **외래 키(Foreign Key, FK)**의 관계를 알아야 한다.



- **Primary Key (PK)**: 테이블에서 각 행을 고유하게 식별하는 값이다. `users` 테이블의 `user_id` 나 `products` 테이블의 `product_id` 가 여기에 해당한다. **PK는 테이블 내에서 절대로 중복될 수 없다.**
- **Foreign Key (FK)**: 다른 테이블의 PK를 참조하는 값이다. `orders` 테이블의 `user_id` 는 `users` 테이블의 `user_id` 를 참조하는 FK다. **FK는 참조하는 테이블에서 여러 번 중복되어 나타날 수 있다.** 예를 들어, 한 명의 고객(`user_id`)이 여러 번의 주문(`orders`)을 할 수 있기 때문이다.

이 관계를 '부모-자식 관계'에 비유하면 이해하기 쉽다.

- **부모 테이블 (Parent Table)**: PK를 가지고 있는 테이블. (`users`, `products`)
- **자식 테이블 (Child Table)**: FK를 통해 부모 테이블을 참조하는 테이블. (`orders`)

이 개념을 바탕으로 조인 시 데이터 행 수가 어떻게 변하는지 정리하면 다음과 같다.

1. **자식 → 부모 조인 (FK → PK 참조)**: 행 개수가 늘어나지 않는다.
    - `orders` 테이블을 기준으로 `users` 테이블을 조인하는 경우다.
    - 자식 테이블(`orders`)의 각 주문 정보는 반드시 **단 한 명의 부모(`users`)**하고만 연결된다. 주문 하나가 여러 고객의 것일 수는 없기 때문이다. 따라서 기준 테이블인 `orders` 의 행 개수가 그대로 유지된다.
    - PK는 유일한 하나의 값만 저장된다. 따라서 PK 방향으로 참조하는 경우 행 개수가 늘어나지 않는다.

2. **부모 → 자식 조인 (PK → FK 참조)**: 행 개수가 늘어날 수 있다.
    - `users` 테이블을 기준으로 `orders` 테이블을 조인하는 경우다.
    - 부모 테이블(`users`)의 한 고객은 **여러 명의 자식(여러 건의 `orders`)**을 가질 수 있다. 이 경우, 한 고객의 정보를 여러 주문 정보에 각각 매칭시켜야 하므로, 고객 정보 행이 주문 건수만큼 복제되어 전체 행의 수가 늘어난다.
    - FK는 같은 값을 여러개 저장할 수 있다. 따라서 FK 방향으로 참조하는 경우 행 개수가 늘어날 수 있다.

말로만 들으면 어려우니, 실제 예제를 통해 직접 확인해 보자. 특정 고객인 '션'(`user_id=1`)의 데이터를 중심으로, 조인의 양방향을 살펴보며 그 원리를 단계별로 파헤쳐 보자.

특정 고객 '션'으로 조인 원리 파헤치기

먼저 `users` 테이블과 `orders` 테이블에서 '션'의 데이터를 확인해 보자.

`users` 테이블에서 '션'(`user_id:1`)의 정보는 단 **한 행**이다. `user_id` 가 PK이므로 유일하게 하나여야 한다.

```
SELECT user_id, name, email
FROM users
WHERE user_id = 1;
```

user_id	name	email
1	션	sean@example.com

`orders` 테이블을 보면 '션'(`user_id=1`)이 주문한 내역은 **두 행**이다. `user_id`가 FK이므로 이렇게 중복이 가능하고, 여러 행이 조회될 수 있다.

```sql
SELECT order_id, product_id, user_id
FROM orders
WHERE user_id = 1;
```

order_id	product_id	user_id
1	1	1
2	4	1

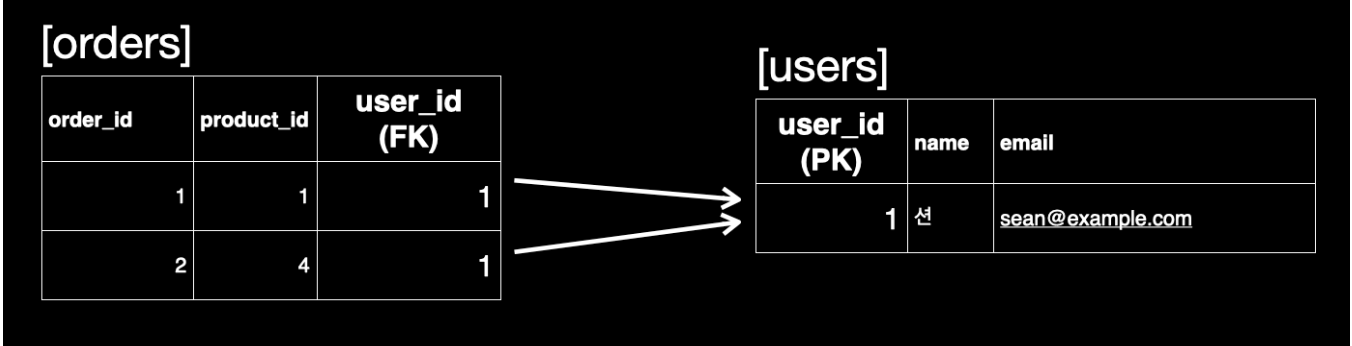
이제 이 두 데이터를 조인해 보자.

경우 1: 자식 → 부모 조인 (orders → users, 행 개수 유지)

먼저 `orders` 테이블을 기준으로 '션'의 주문 내역에 고객 이름을 붙여보자. 이것이 바로 **자식 테이블(`orders`)에서 부모 테이블(`users`)로, 즉 FK → PK로 조인**하는 경우이다.

```sql
SELECT
    o.order_id,
    o.product_id,
    o.user_id AS orders_user_id,
    u.user_id AS users_user_id,
    u.name,
    u.email
FROM
    orders o
JOIN
    users u ON o.user_id = u.user_id
WHERE
    o.user_id = 1;
```

[조인 과정]



- 기준 테이블인 `orders`에는 '션'의 주문이 2행 있다(`order_id: 1`, `order_id: 2`).
- 첫 번째 주문(`order_id: 1`)의 `user_id`는 1이다. 이 값은 `users` 테이블에서 PK가 1인 '션'의 **단일 행**과 매칭된다.
- 두 번째 주문(`order_id: 2`)의 `user_id` 역시 1이다. 이 값도 `users` 테이블에서 PK가 1인 '션'의 **동일한 단일 행**과 매칭된다.
- 각 주문 행이 오직 하나의 고객 행과 연결되므로, 기준 테이블(`orders`)의 행 수가 그대로 유지된다.

[조인 결과]

order_id	product_id	orders_user_id	users_user_id	name	email
1	1	1	1	션	sean@example.com
2	4	1	1	션	sean@example.com

orders 영역 users 영역

[실행 결과]

order_id	product_id	orders_user_id	users_user_id	name	email
1	1	1	1	션	sean@example.com
2	4	1	1	션	sean@example.com

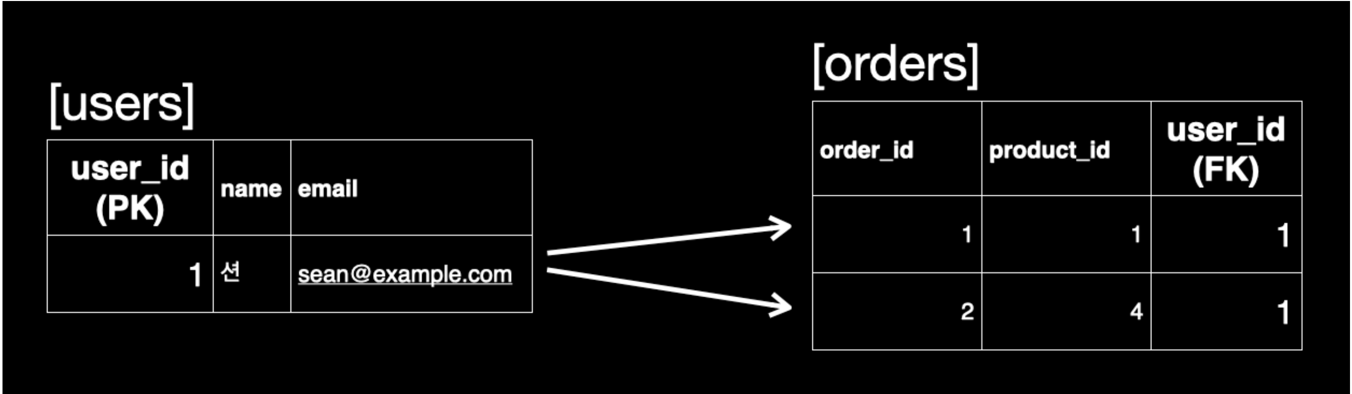
`orders` 테이블은 조인 전에도 2행이었고, 조인 이후에도 2행이 그대로 유지되었다. 이것이 자식 → 부모 조인의 특징이다.

경우 2: 부모 → 자식 조인 (users → orders, 행 개수 증가)

이번에는 반대로 `users` 테이블을 기준으로 '션' 고객의 주문 내역을 붙여보자. 이것이 바로 **부모 테이블(`users`)에서 자식 테이블(`orders`)로, 즉 PK → FK로 조인**하는 경우이다.

```
SELECT
    u.user_id AS users_user_id,
    u.name,
    u.email,
    o.order_id,
    o.product_id,
    o.user_id AS orders_user_id
FROM
    users u
JOIN
    orders o ON u.user_id = o.user_id
WHERE
    u.user_id = 1;
```

[조인 과정]



- 기준 테이블인 `users`에는 '션'의 정보가 **단 1행** 있다.
- 이 '션'의 `user_id: 1`과 일치하는 행을 `orders` 테이블에서 찾는다.
- `orders` 테이블에는 `user_id`가 1인 주문이 **2개**(`order_id: 1`, `order_id: 2`) 있다.
- `users`의 '션' 정보 1행이 `orders`의 2개 행과 관계를 맺어야 한다.
- 데이터베이스는 관계를 표현하기 위해 '션'의 정보 행을 **복제**하여 각 주문에 맞추어 하나씩 붙인다.
    - '션' 정보 (복제본 1) + 첫 번째 주문 정보

- '션' 정보 (복제본 2) + 두 번째 주문 정보
- 결과적으로 1개의 행이 2개의 행으로 늘어난다.

[조인 결과]

users_user_id	name	email	order_id	product_id	orders_user_id
1	션	sean@example.com	1	1	1
1	션	sean@example.com	2	4	1

users 영역 orders 영역

[실행 결과]

users_user_id	name	email	order_id	product_id	orders_user_id
1	션	sean@example.com	1	1	1
1	션	sean@example.com	2	4	1

결과를 보자. 이번에도 결과는 2행이다. 하지만 **기준으로 삼았던** `users` **테이블에서 '션'의 정보는 원래 1행이었다.** 그런데 조인의 결과는 2행이 되었다. 바로 이 지점이 `JOIN`으로 인해 행이 증가하는 핵심 원리다. 부모 테이블의 한 행이 자식 테이블의 여러 행과 관계를 맺을 때, 그 관계의 수만큼 행이 복제되어 결과가 늘어나는 것이다.

전체 데이터로 확장하기

'션'의 예시를 통해 원리를 이해했으니, 이제 `WHERE` 절을 제거하고 전체 데이터를 대상으로 조인 결과를 살펴보자.

먼저 `users` 테이블과 `orders` 테이블에서 전체 데이터를 확인해 보자.

`users` 테이블 - 6행

```
SELECT user_id, name, email
FROM users;
```

[실행 결과]

user_id	name	email
1	션	sean@example.com
2	네이트	nate@example.com
3	세종대왕	sejong@example.com
4	이순신	sunsin@example.com
5	마리 퀴리	marie@example.com
6	레오나르도 다빈치	vinci@example.com

`orders` 테이블 - 7행

```
SELECT order_id, product_id, user_id
FROM orders;
```

[실행 결과]

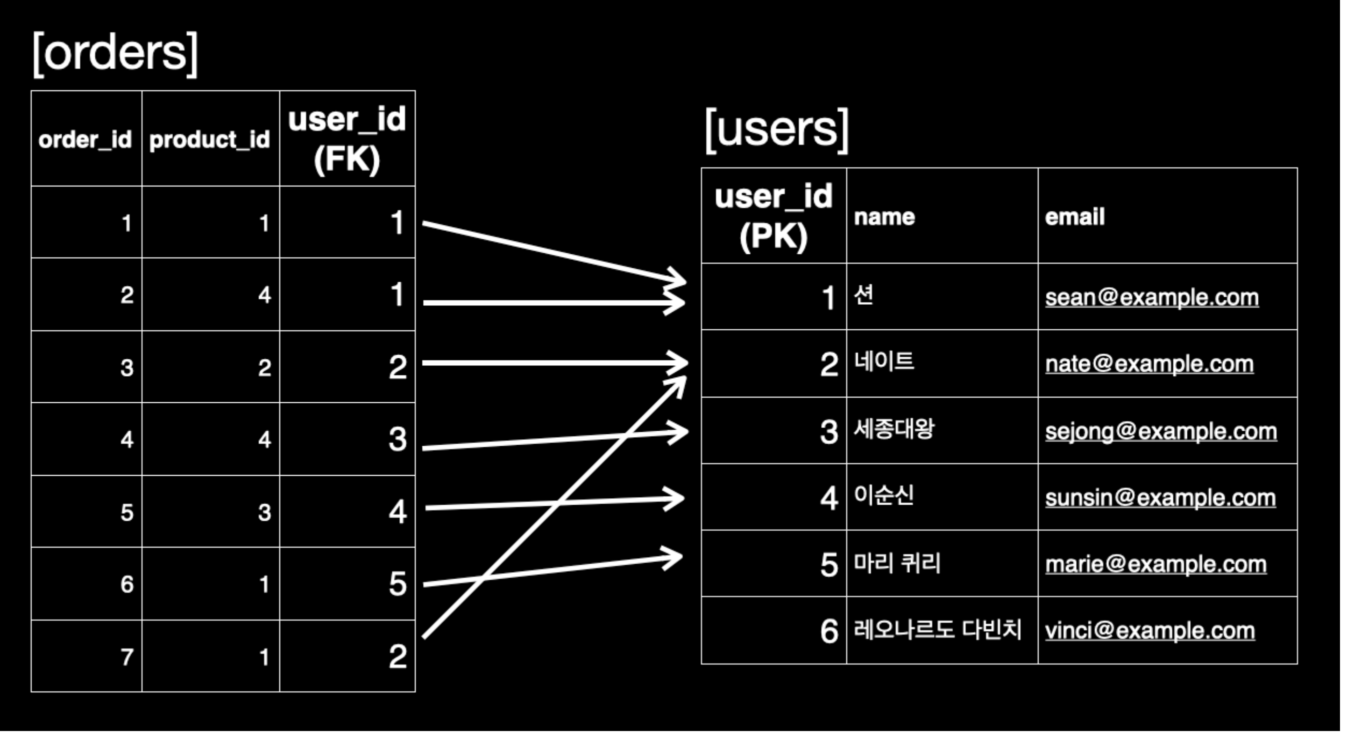
order_id	product_id	user_id
1	1	1
2	4	1
3	2	2
4	4	3
5	3	4
6	1	5
7	1	2

전체 주문(Orders) 기준 (행 개수 유지)

`orders` 테이블(7행)을 기준으로 모든 주문에 고객 이름을 붙여보자. **자식 → 부모** 조인이다.

```
SELECT
    o.order_id,
    o.product_id,
    o.user_id AS orders_user_id,
    u.user_id AS users_user_id,
    u.name,
    u.email
FROM
    orders o
JOIN
    users u ON o.user_id = u.user_id;
```

[조인 과정]



- `orders` 테이블의 각 행은 FK인 `user_id` 를 가지고 있다.
- 이 FK는 `users` 테이블의 PK인 `user_id` 를 참조한다.
- PK는 고유하므로, `orders` 테이블의 각 행은 언제나 `users` 테이블의 **단 하나의 행**하고만 연결된다.

[실행 결과]

order_id	product_id	orders_user_id	users_user_id	name	email

1	1	1	1	션	sean@example.com
2	4	1	1	션	sean@example.com
3	2	2	2	네이트	nate@example.com
7	1	2	2	네이트	nate@example.com
4	4	3	3	세종대왕	sejong@example.com
5	3	4	4	이순신	sunsin@example.com
6	1	5	5	마리 퀴리	marie@example.com

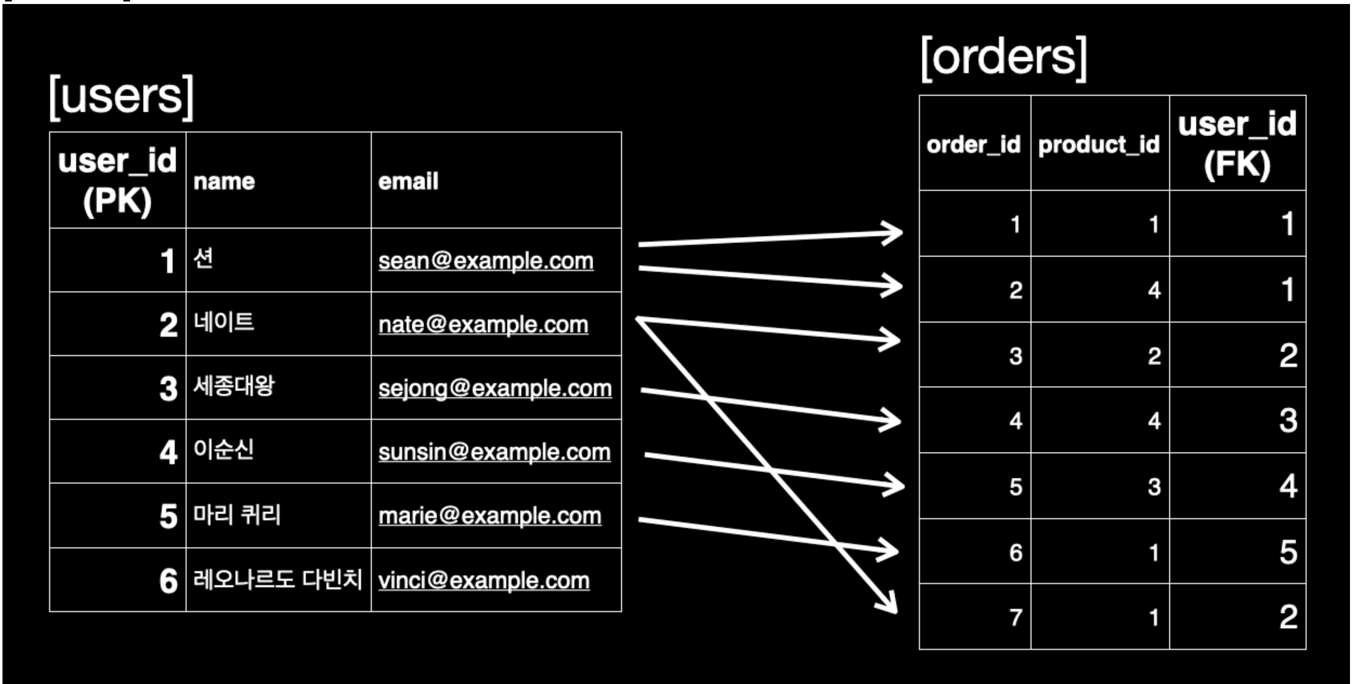
예상대로 결과는 7행이다. 기준 테이블인 `orders` 의 7개 주문이 각각 단 한 명의 고객과 연결되므로, 행의 개수는 그대로 유지된다.

전체 고객(Users) 기준 (행 개수 증가)

`users` 테이블(6행)을 기준으로 모든 고객의 주문 내역을 붙여보자. **부모 → 자식** 조인이다.

```
SELECT
    u.user_id AS users_user_id,
    u.name,
    u.email,
    o.order_id,
    o.product_id,
    o.user_id AS orders_user_id
FROM
    users u
JOIN
    orders o ON u.user_id = o.user_id
ORDER BY users_user_id;
```

[조인 과정]



[실행 결과]

users_user_id	name	email	order_id	product_id	orders_user_id
1	션	sean@example.com	1	1	1
1	션	sean@example.com	2	4	1
2	네이트	nate@example.com	3	2	2
2	네이트	nate@example.com	7	1	2
3	세종대왕	sejong@example.com	4	4	3
4	이순신	sunsin@example.com	5	3	4
5	마리 퀴리	marie@example.com	6	1	5

결과는 7행이다. `users` 테이블은 6행이었지만, 조인 결과는 7행이 되었다.
'션'(`user_id: 1`)과 '네이트'(`user_id: 2`)가 각각 2건의 주문을 가지고 있었기 때문에, 이들의 정보가 한 번씩 더 복제되어 총 7개의 행이 만들어졌다.
참고로 주문이 없는 '레오나르도 다빈치'(`user_id: 6`)는 `INNER JOIN`의 특성상 `orders` 테이블에 매칭되는 데이터가 없으므로 결과에서 제외되었다. 만약 `OUTER JOIN`을 사용했다면 결과에 포함되었을 것이다.

정리: 언제 행이 늘어나고 언제 그대로인가?

지금까지의 내용을 정리해 보자. 조인 시 결과 행의 수가 변하는지 여부는 두 테이블의 관계와 어떤 테이블을 기준으로 삼는지에 달려있다.

- **행 개수 유지: 자식에서 부모로 조인할 때 (`to-One`)**
  - **FK ➔ PK 조인**
  - **방향**: `FROM 자식 테이블 JOIN 부모 테이블` (예: `FROM orders JOIN users`)
  - **원리**: 자식 테이블의 모든 행은 부모 테이블의 **단 하나의 행**과 매칭된다. (주문은 반드시 한 명의 고객에게 속한다.)
  - **결과**: 기준이 되는 자식 테이블의 행 개수가 그대로 유지된다. `orders` 테이블이 7행이면 결과도 7행이다.

- **행 개수 증가 가능: 부모에서 자식으로 조인할 때 (`to-Many`)**
  - **PK ➔ FK 조인**
  - **방향**: `FROM 부모 테이블 JOIN 자식 테이블` (예: `FROM users JOIN orders`)
  - **원리**: 부모 테이블의 한 행은 자식 테이블의 **여러 행**과 매칭될 수 있다. (한 명의 고객이 여러 번 주문할 수 있다.)
  - **결과**: 부모 행이 자식 행의 개수만큼 **복제**되면서 전체 행의 수가 기준 테이블보다 늘어날 수 있다. 주문을 2번 한 고객은 결과 테이블에서 2개의 행을 차지하게 된다.

실무에서 이것이 왜 중요할까?

이 원리를 모르면 데이터를 잘못 분석하게 될 위험이 크다. 예를 들어, 모든 고객과 그들의 주문 정보를 보기 위해 `FROM users JOIN orders`를 수행했다고 하자. 이 결과에서 고객 수를 세기 위해 `COUNT(u.user_id)`를 실행하면 어떻게 될까? 전체 고객 수인 6이 나올까? 아니다. 주문을 여러 번 한 고객이 중복 계산되므로, 전체 주문 수인 7이 나온다.

이처럼 조인으로 인해 데이터가 어떻게 변하는지 정확히 이해해야만, 합계(`SUM`), 평균(`AVG`), 개수(`COUNT`) 같은 집계 함수를 올바르게 사용하고 원하는 분석 결과를 정확하게 도출할 수 있다. 쿼리를 작성하기 전에 항상 **어떤 테이블을 기준으로 삼을지**, 그리고 조인으로 인해 행 수가 증가하는 상황인지 아닌지, 먼저 생각하는 습관을 들이는 것이 중요하

다.

참고: 조인의 유연성

데이터베이스에서 조인(JOIN)은 주로 **기본 키(PK)와 외래 키(FK)를 써서 테이블을 연결하는 것이 가장 일반적이고 중요한 방식**이다. 그러나 조인의 핵심 원리는 '두 테이블의 특정 열(column)의 값이 같은가?' 이기에, 실제로는 어떤 열이든 조인 조건으로 쓸 수 있다. 데이터 타입만 같다면 말이다.

다양한 조인 예시

PK-FK 관계가 아니더라도 여러 상황에서 조인을 활용할 수 있다.

동명이인 찾기 (이름으로 조인)

고객 테이블과 직원 테이블에 모두 '이름' 열이 있다면, 이 두 테이블을 이름으로 조인하여 고객과 직원의 이름이 같은 경우를 찾아낸다.

```
SELECT A.이름, A.연락처, B.부서
FROM 고객 AS A
JOIN 직원 AS B ON A.이름 = B.이름;
```

특정 날짜의 이벤트 연결 (날짜로 조인)

주문 테이블의 '주문일자'와 마케팅_이벤트 테이블의 '이벤트_시작일'을 기준으로 조인하여, 특정 이벤트가 있던 날에 들어온 주문을 분석할 수 있다.

```
SELECT A.주문번호, A.주문금액, B.이벤트명
FROM 주문 AS A
JOIN 마케팅_이벤트 AS B ON A.주문일자 = B.이벤트_시작일;
```

지역별 데이터 분석 (지역 코드로 조인)

매장 테이블의 '우편번호' 앞 두 자리와 지역별_인구통계 테이블의 '지역코드'를 연결하여, 매장이 있는 지역의 인구 통계 데이터를 함께 분석한다.

```
SELECT A.매장명, B.평균소득
FROM 매장 AS A
JOIN 지역별_인구통계 AS B ON SUBSTRING(A.우편번호, 1, 2) = B.지역코드;
```

로그 데이터 분석 (상태 코드로 조인)

웹서버_로그 테이블의 '상태코드'와 에러_코드_정의 테이블의 '코드'를 조인하여, 로그에 기록된 에러가 무엇을 뜻하는지 바로 파악할 수 있다.

```
SELECT A.요청URL, B.에러설명
FROM 웹서버_로그 AS A
JOIN 에러_코드_정의 AS B ON A.상태코드 = B.코드;
```

PK-FK 조인이 중요한 이유

이처럼 조인은 매우 유연하지만, 실무에서는 데이터의 정확성과 일관성을 위해 대부분 PK와 FK를 쓴다. 이름처럼 중복될 수 있거나, 언제든 바뀔 수 있는 값으로 조인하면 데이터가 엉뚱하게 연결될 위험이 크기 때문이다.

안정적인 PK-FK 관계를 이해하는 것이야말로 조인을 제대로 활용하는 첫걸음이다.

이제 테이블을 연결하는 기본적인 방법들은 배웠다. 그런데 만약 연결할 대상이 다른 테이블이 아니라 '자기 자신'이라면 어떻게 해야 할까? 다음 시간에는 이처럼 자기 자신을 참조하는 특별한 조인, `SELF JOIN`에 대해 알아보겠다.

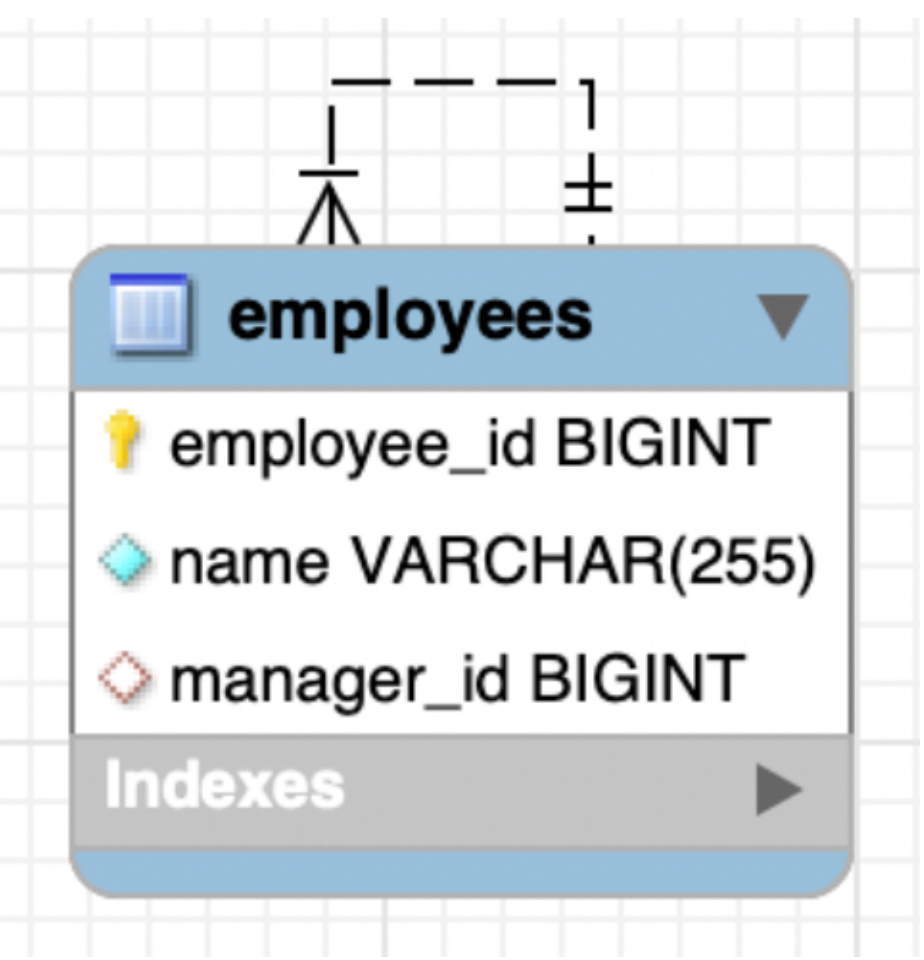
셀프 조인

우리는 지금까지 서로 다른 테이블, 즉 `users`와 `orders`처럼 명확하게 구분된 대상을 연결하는 법을 배웠다. 그런데 만약 연결해야 할 대상이 다른 테이블이 아니라 바로 '자기 자신'이라면 어떻게 해야 할까?

실무에서는 이런 경우가 생각보다 흔하다. 우리 회사 조직도를 관리하는 `employees` 테이블을 예로 들어보자. 이 테이블에는 모든 직원의 정보가 들어있다. 그런데 각 직원의 '상사' 또한 우리 회사의 '직원'이다. 즉, 상사 정보 역시 `employees` 테이블 안에 들어있다.

여기서 오늘의 문제 상황이 나온다.
"각 직원의 이름과 바로 위 직속 상사의 이름을 나란히 함께 출력하려면 어떻게 해야 할까?"

먼저 `employees` 테이블의 구조를 살펴보자.




```
SELECT * FROM employees;
```

[실행 결과]

employee_id	name	manager_id
1	김회장	NULL
2	박사장	1
3	이부장	2
4	최과장	3
5	정대리	4
6	홍사원	4

'홍사원'(`employee_id: 6`)의 `manager_id`는 4다. 이 상사의 이름을 알려면, 우리는 다시 이 테이블에서 `id`가 4

인 직원을 찾아야 한다. '최과장'이라는 것을 알 수 있다. 이처럼 한 테이블 안에서 자신의 컬럼(`manager_id`)이 같은 테이블의 다른 컬럼(`employee_id`)을 참조하는 구조를 '자기 참조 관계'라고 한다.

이런 구조의 데이터를 한 번의 쿼리로 조회하기 위해 사용하는 기술이 바로 `SELF JOIN`이다.

SELF JOIN의 개념과 원리

`SELF JOIN`은 `INNER JOIN`, `OUTER JOIN`처럼 새로운 종류의 `JOIN` 명령어가 아니다. 이것은 **하나의 테이블을 자기 자신과 조인하는 '기법'**을 일컫는 말이다.

SQL이 이 기법을 가능하게 하는 핵심 원리는 바로 **테이블 별칭(Alias)**에 있다. 하나의 테이블에 서로 다른 별칭을 두 개 부여함으로써, 데이터베이스가 이들을 마치 다른 두 개의 테이블인 것처럼 인식하게 만드는 것이다.

우리는 `employees` 테이블을 두 개 복사해서 하나는 직원을 나타내는 `e` (employee)로, 다른 하나는 상사를 나타내는 `m` (manager)으로 사용한다고 상상하면 이해하기 쉽다.

- `e` 테이블: 모든 직원의 목록
- `m` 테이블: 모든 상사의 목록 (실체는 똑같은 `employees` 테이블)

그리고 `e` 테이블의 `manager_id`와 `m` 테이블의 `employee_id`가 같은 것들을 연결(`JOIN`)해주면, 우리는 직원의 이름(`e.name`)과 상사의 이름(`m.name`)을 한 줄에 나란히 놓을 수 있게 된다.

실습: 직원-상사 목록 만들기

이제 `SELF JOIN` 기법을 사용해서 직원과 상사 목록을 만드는 쿼리를 작성해 보자.

1단계: `INNER JOIN`을 이용한 SELF JOIN

가장 기본적인 `INNER JOIN`으로 두 가상 테이블 `e`와 `m`을 연결한다. 연결 조건은 "직원의 매니저 ID(`e.manager_id`)가 상사의 직원 ID(`m.employee_id`)와 같을 때"이다.

```sql
SELECT
    e.name AS employee_name,
    m.name AS manager_name
FROM
    employees e
JOIN
    employees m ON e.manager_id = m.employee_id;
```

`e.name`은 직원의 이름, `m.name`은 상사의 이름이 된다. 컬럼에도 `AS`를 사용해 별칭을 붙여주면 결과를 이해하기가 훨씬 수월하다. (물론 `AS`는 생략할 수 있다.)

[조인 과정]

[employees e]

employee_id	name	manager_id (FK)
1	김회장	NULL
2	박사장	1
3	이부장	2
4	최과장	3
5	정대리	4
6	홍사원	4

[employees m]

employee_id (PK)	name	manager_id
1	김회장	NULL
2	박사장	1
3	이부장	2
4	최과장	3
5	정대리	4
6	홍사원	4

[실행 결과]

employee_name	manager_name
박사장	김회장
이부장	박사장
최과장	이부장
정대리	최과장
홍사원	최과장

결과를 보면, 각 직원 옆에 직속 상사의 이름이 정확히 출력된 것을 확인할 수 있다.

그런데 한 가지 이상한 점이 있다. 전체 직원 중 '김회장'이 `employee_name` 목록에 보이지 않는다. 왜일까? 그의 `manager_id`는 `NULL`이기 때문이다. `INNER JOIN`은 `ON` 조건이 맞는, 즉 `e.manager_id`에 값이 있는 데이터만 결과에 포함시키므로 `manager_id`가 `NULL`인 '김회장'은 조인 대상에서 제외된 것이다.

2단계: `LEFT JOIN`을 이용한 SELF JOIN

만약 상사가 없는 최상위 리더, 즉 '김회장'까지 포함한 전체 직원 목록을 보고 싶다면 어떻게 해야 할까?

이럴 때 바로 `LEFT JOIN`이 필요하다. 직원을 나타내는 `e` 테이블을 기준으로 삼고, 상사 정보를 왼쪽에 붙이는 것이다. 상사 정보가 없으면(`manager_id`가 `NULL`이면) 그 자리는 `NULL`로 표시될 것이다.

```
SELECT
    e.name AS employee_name,
    m.name AS manager_name
FROM
    employees AS e
LEFT JOIN
    employees AS m ON e.manager_id = m.employee_id;
```

[실행 결과]

employee_name	manager_name
김회장	NULL
박사장	김회장
이부장	박사장
최과장	이부장
정대리	최과장
홍사원	최과장

이제 `LEFT JOIN`을 통해 상사가 없는 '김회장'까지 포함된 완벽한 조직도 리스트를 만들었다.

이처럼 `SELF JOIN`은 테이블 별칭을 활용하여 자기 참조 관계를 풀어내는 유용한 기법이다. 조직도뿐만 아니라 웹사이트의 카테고리와 서브카테고리, 게시판의 원본 글과 답변 글 같은 계층형 데이터를 다룰 때 반드시 필요한 기술이니 기억해두자.

지금까지 우리는 테이블에 '이미 존재하는' 데이터들을 다양한 방식으로 연결하는 법을 배웠다. 그런데 만약, 모든 경우

의 수를 조합해서 '존재하지 않는 가상의 데이터' 목록을 만들어야 한다면 어떻게 해야 할까?

예를 들어, 우리 쇼핑몰에서 판매할 티셔츠의 모든 색상과 사이즈 조합(빨강-S, 빨강-M, 파랑-S, 파랑-M...)을 상품 마스터 데이터로 미리 생성해야 한다면? 다음 시간에는 이 문제를 해결해 줄 `CROSS JOIN`에 대해 알아보겠다.

CROSS 조인

지금까지 우리가 배운 `INNER`, `OUTER`, `SELF` 조인은 모두 `ON`이라는 연결고리를 통해 테이블에 '이미 존재하는' 관계를 찾아내는 작업이었다. 즉, 특정 조건에 맞는 짝을 찾는 것이 핵심이었다.

그런데 만약, 애초에 짝이나 관계가 없는 두 집단을 가지고 가능한 모든 조합을 만들어내야 한다면 어떻게 해야 할까?

오늘의 문제 상황은 바로 이것이다.
"우리 쇼핑몰에서 새로운 기본 티셔츠를 판매하려고 한다. 사이즈는 `S, M, L, XL` 네 종류이고, 색상은 `Red, Blue, Black` 세 종류이다. 판매를 시작하기 전에, 재고 관리를 위해 가능한 모든 사이즈와 색상 조합을 담은 상품 마스터 데이터를 미리 생성하고 싶다."

이 업무는 'S 사이즈이면서 Red 색상인 상품', 'M 사이즈이면서 Red 색상인 상품' ... 'XL 사이즈이면서 Black 색상인 상품' 까지, 생각할 수 있는 모든 경우의 수를 목록으로 만들어야 한다. `sizes` 테이블과 `colors` 테이블 사이에는 미리 정해진 연결고리(`ON` 조건)가 없다. 우리는 이 둘을 강제로 조합해야 한다.

이럴 때 사용하는 조인이 바로 `CROSS JOIN`이다.

CROSS JOIN의 개념과 카테시안 곱 (Cartesian Product)

`CROSS JOIN`은 조인 조건 없이, 한쪽 테이블의 모든 행을 다른 쪽 테이블의 모든 행과 하나씩 전부 연결하는, 가장 단순한 조인 방식이다. 연결고리가 없기 때문에 `ON` 절을 사용하지 않는다.

`CROSS JOIN`의 결과를 수학 용어로 **카테시안 곱(Cartesian Product)** 또는 데카르트 곱이라고 부른다. 만약 A 테이블에 `m`개의 행이 있고, B 테이블에 `n`개의 행이 있다면, 두 테이블을 `CROSS JOIN`한 결과는 `m * n`개의 행을 갖게 된다.

우리 예시의 `sizes` 테이블은 4개의 행을, `colors` 테이블은 3개의 행을 가지고 있다. 따라서 두 테이블을 `CROSS

`JOIN`하면 `4 * 3 = 12`개의 행으로 이루어진 결과가 나올 것이다.

실습: 상품 옵션 조합 만들기

이제 `CROSS JOIN`을 사용해 티셔츠의 모든 사이즈와 색상 조합을 만들어 보자.

먼저 실습에 사용할 `sizes` 테이블과 `colors` 테이블의 데이터를 확인하자.

```sql
SELECT * FROM sizes;
```

size
S
M
L
XL

```sql
SELECT * FROM colors;
```

color
Black
Blue
Red

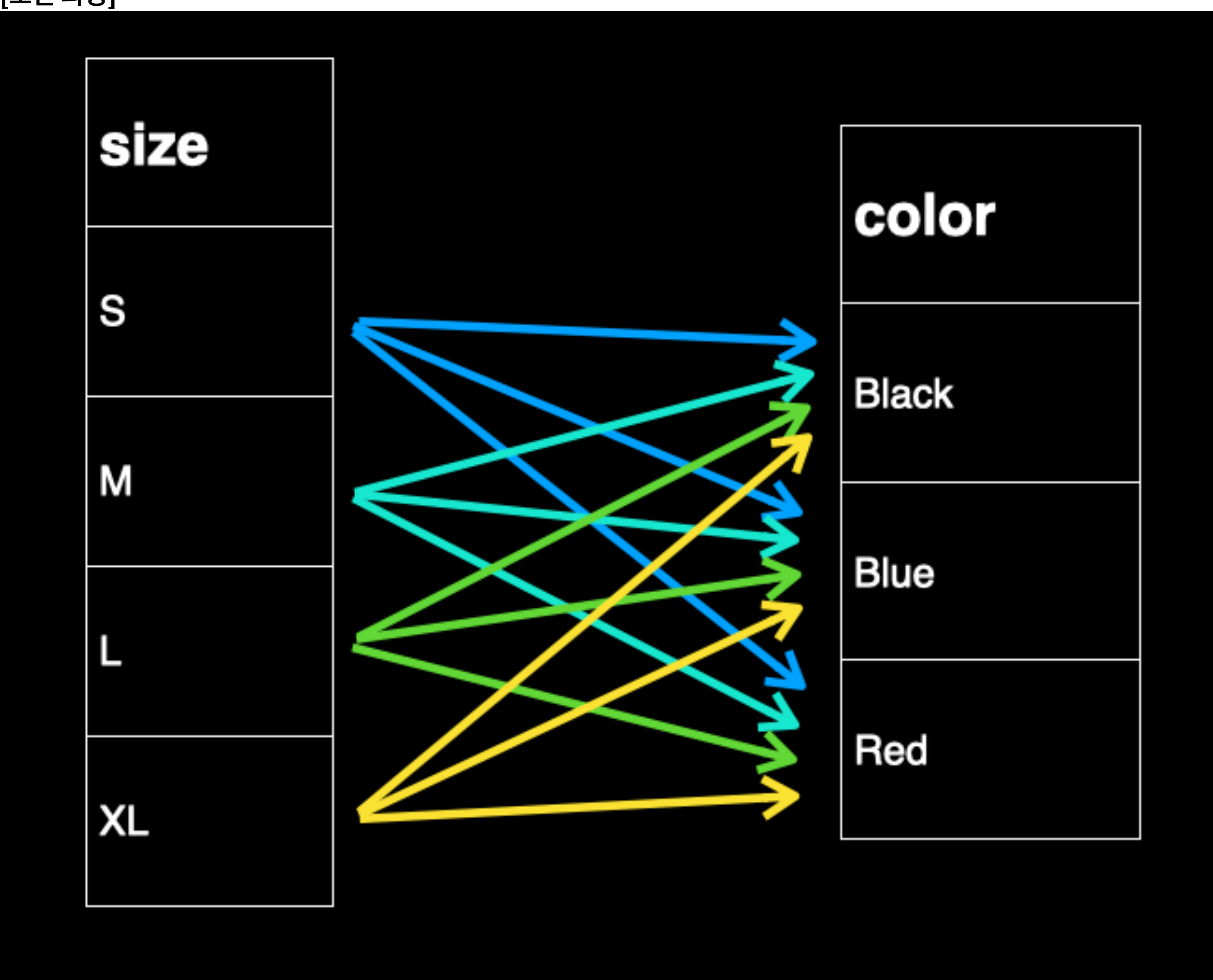
이제 두 테이블을 `CROSS JOIN`하여 모든 조합을 생성하자.

```sql
SELECT
    s.size,
    c.color
```

```
FROM
    sizes s
CROSS JOIN
    colors c;
```

- `ON` 절이 없는 것을 다시 한번 확인하자.

[조인 과정]



[실행 결과]

size	color
S	Red
S	Blue
S	Black

M	Red
M	Blue
M	Black
L	Red
L	Blue
L	Black
XL	Red
XL	Blue
XL	Black

보다시피 `sizes` 테이블의 각 행이 `colors` 테이블의 모든 행과 하나씩 짝을 이뤄 총 4 x 3, 12개의 완벽한 조합이 만들어졌다.

여기서 한 걸음 더 나아가, 이 조합을 이용해 상품명을 자동으로 생성해 볼 수도 있다. 문자열을 합치는 `CONCAT` 함수를 활용하면 된다.

```
SELECT
    CONCAT('기본티셔츠-', c.color, '-', s.size) AS product_name,
    s.size,
    c.color
FROM
    sizes AS s
CROSS JOIN
    colors AS c;
```

[실행 결과]

product_name	size	color
기본티셔츠-Red-S	S	Red
기본티셔츠-Blue-S	S	Blue

기본티셔츠-Black-S	S	Black
기본티셔츠-Red-M	M	Red
기본티셔츠-Blue-M	M	Blue
기본티셔츠-Black-M	M	Black
기본티셔츠-Red-L	L	Red
기본티셔츠-Blue-L	L	Blue
기본티셔츠-Black-L	L	Black
기본티셔츠-Red-XL	XL	Red
기본티셔츠-Blue-XL	XL	Blue
기본티셔츠-Black-XL	XL	Black

이 결과를 새로운 테이블에 `INSERT` 하면, 상품 마스터 데이터를 손쉽게 구축할 수 있다.

`INSERT INTO ... SELECT` 로 상품 옵션 마스터 테이블 만들기

이제 `CROSS JOIN` 으로 만든 상품 옵션 조합을 새로운 테이블에 저장해 보자. 이렇게 데이터를 한 번에 대량으로 삽입할 때 `INSERT INTO ... SELECT` 구문을 사용한다.

우리는 티셔츠 상품에 대한 모든 사이즈와 색상 조합을 저장할 `product_options` 테이블을 만들 것이다. 이 테이블은 `option_id`, `product_name`, `size`, `color` 필드를 가진다.

먼저, `product_options` 테이블을 생성한다.

```
CREATE TABLE product_options (
    option_id BIGINT AUTO_INCREMENT,
    product_name VARCHAR(255) NOT NULL,
    size VARCHAR(10) NOT NULL,
    color VARCHAR(20) NOT NULL,
    PRIMARY KEY (option_id)
);
```

테이블이 준비되었으니, 이제 `CROSS JOIN`의 결과를 `product_options` 테이블에 `INSERT` 하자.

```
INSERT INTO product_options (product_name, size, color)
SELECT
    CONCAT('기본티셔츠-', c.color, '-', s.size) AS product_name,
    s.size,
    c.color
FROM
    sizes AS s
CROSS JOIN
    colors AS c;
```

`INSERT INTO ... SELECT`는 `SELECT` 문으로 조회된 결과를 즉시 다른 테이블에 삽입하는 기능이다. 이 방법을 사용하면 대량의 초기 데이터를 효율적으로 구축하거나, 기존 데이터를 가공하여 새로운 테이블에 저장하는 등의 작업을 할 수 있다.

이제 `product_options` 테이블에 데이터가 잘 들어갔는지 확인해 보자.

```
SELECT * FROM product_options;
```

[실행 결과]

option_id	product_name	size	color
1	기본티셔츠-Red-S	S	Red
2	기본티셔츠-Blue-S	S	Blue
3	기본티셔츠-Black-S	S	Black
4	기본티셔츠-Red-M	M	Red
5	기본티셔츠-Blue-M	M	Blue
6	기본티셔츠-Black-M	M	Black
7	기본티셔츠-Red-L	L	Red
8	기본티셔츠-Blue-L	L	Blue

9	기본티셔츠-Black-L	L	Black
10	기본티셔츠-Red-XL	XL	Red
11	기본티셔츠-Blue-XL	XL	Blue
12	기본티셔츠-Black-XL	XL	Black

`CROSS JOIN`으로 생성한 12개의 상품 옵션 조합이 `product_options` 테이블에 성공적으로 삽입된 것을 확인할 수 있다.

이처럼 `CROSS JOIN`은 단순히 데이터를 조합하는 것을 넘어, 새로운 데이터 세트를 만들거나 초기 시스템을 구축할 때 유용하게 활용될 수 있다. 특히 상품의 다양한 옵션(사이즈, 색상, 재질 등)을 조합하여 마스터 데이터를 생성하는 시나리오에서 좋은 도구가 된다.

실무에서의 치명적인 주의사항

`CROSS JOIN`은 모든 경우의 수를 만들어주기 때문에 유용하지만, 실무에서는 아주 신중하게 사용해야 하는, 어떻게 보면 가장 위험한 조인이기도 하다.

왜냐하면 결과의 행 수가 기하급수적으로 늘어날 수 있기 때문이다. 만약 당신이 실수로 100만 건의 데이터가 있는 `users` 테이블과 10만 건의 데이터가 있는 `products` 테이블을 `CROSS JOIN` 한다면 어떻게 될까? 결과는 `100만 * 10만 = 1000억` 건이 된다. 이 쿼리를 실행하는 순간 데이터베이스 서버는 아마 응답을 멈추거나 다운될 것이다.

지금까지 `INNER`, `OUTER`, `SELF`, 그리고 `CROSS JOIN`까지, 조인의 거의 모든 것을 배웠다.
이제 이 도구들을 모두 꺼내서, 실무에서 마주칠 법한 조금 더 복잡한 요구사항을 해결해 볼 시간이다. 여러 테이블을 동시에 조인하고, 조건을 걸고, 정렬하는 종합 실습을 통해 `JOIN`을 완벽하게 우리 것으로 만들어 보자.

조인 종합 실습

지금까지 우리는 `JOIN`이라는 강력한 무기를 손에 넣기 위해 `INNER`, `OUTER`, `SELF`, `CROSS JOIN`을 각각 연마했다. 이제는 이 모든 기술을 종합적으로 활용하여, 실무에서 마주칠 법한 복잡한 요구사항을 해결해 볼 시간이다.

하나의 `JOIN`만으로는 풀 수 없는, 여러 테이블과 여러 조건이 얽혀있는 문제를 마주했을 때, 당황하지 않고 차근차근 해결해 나가는 능력을 기르는 것이 이번 수업의 목표다.

오늘 우리가 해결해야 할 최종 미션은 다음과 같다.
"2025년 6월에 '서울'에 거주하는 고객이 주문한 모든 내역에 대해, `고객 이름`**,** `고객 이메일`**,** `주문 날짜`**,** `주문한 상품명`**,** `상품 가격`**,** `주문 수량`**을 포함하는 상세 보고서를 최신 주문 순으로 작성하라."**

문제 분석 및 해결 전략

문제가 길고 복잡해 보인다. 이럴 때는 요구사항을 잘게 쪼개서 하나씩 살펴보는 것이 최고의 전략이다.

1. **필요한 정보는 무엇인가?**
   - 고객 이름, 고객 이메일 ➔ `users` 테이블
   - 주문 날짜, 주문 수량 ➔ `orders` 테이블
   - 주문한 상품명, 상품 가격 ➔ `products` 테이블
   - 주문 내역에 대한 것이므로 주문 테이블을 기준으로 시작하는 것이 좋다.

2. **필요한 테이블은 무엇인가?**
   - 위 정보를 모두 얻으려면 `users`, `orders`, `products` 세 개의 테이블이 모두 필요하다.

3. **필터링 조건은 무엇인가?**
   - 조건 1: 2025년 6월에 이루어진 주문 (`orders` 테이블)
   - 조건 2: '서울'에 거주하는 고객의 주문 (`users` 테이블)

4. **정렬 순서는 무엇인가?**
   - 최신 주문 순서 (`orders` 테이블의 `order_date` 기준 내림차순)

전략이 명확해졌다. `users`, `orders`, `products` 세 테이블을 모두 조인한 후, `WHERE` 절로 두 가지 조건을 필터링하고, 최종 결과를 `ORDER BY`로 정렬하면 된다.

단계별 쿼리 작성

이제 차근차근 쿼리를 완성해 보자.

1단계: `orders`와 `users` 조인하기

가장 먼저 주문과 고객을 연결한다. 주문 기록이 있는 고객만 필요하므로 `INNER JOIN`이 적합하다.

```
SELECT
    u.user_id, u.name, u.email, u.address, -- 고객
    o.order_id, o.order_date, o.quantity -- 주문
FROM orders o
JOIN users u ON o.user_id = u.user_id;
```

user_id	name	email	address	order_id	order_date	quar
1	션	sean@example.com	서울시 강남구	1	2025-06-10 10:00:00	1
1	션	sean@example.com	서울시 강남구	2	2025-06-10 10:05:00	2
2	네이트	nate@example.com	경기도 성남시	3	2025-06-11 14:20:00	1
3	세종대왕	sejong@example.com	서울시 종로구	4	2025-06-12 09:00:00	1
4	이순신	sunsin@example.com	전라남도 여수시	5	2025-06-15 11:30:00	1
5	마리 퀴리	marie@example.com	서울시 강남구	6	2025-06-16 18:00:00	1
2	네이트	nate@example.com	경기도 성남시	7	2025-06-17 12:00:00	2

- 고객 테이블과 주문 테이블을 조인을 사용해서 연결했다.

2단계: `products` 테이블 추가로 조인하기

위에서 조인된 결과에 이제 상품 정보를 연결한다. `orders` 테이블의 `product_id` 와 `products` 테이블의 `product_id` 를 연결고리로 사용한다.

```
SELECT
    u.user_id, u.name, u.email, u.address, -- 고객
    o.order_id, o.order_date, o.quantity, -- 주문
    p.product_id, p.name, p.price -- 상품
FROM orders o
```

```
JOIN users u ON o.user_id = u.user_id
JOIN products p ON o.product_id = p.product_id
```

user_id	name	...	order_id	...	product_id	name	price
1	션	...	1	...	1	프리미엄 게이밍 마우스	7500(
1	션	...	2	...	4	관계형 데이터베이스 입문	2800(
2	네이트	...	3	...	2	기계식 키보드	1200(
3	세종대왕	...	4	...	4	관계형 데이터베이스 입문	2800(
4	이순신	...	5	...	3	4K UHD 모니터	3500(
5	마리 퀴리	...	6	...	1	프리미엄 게이밍 마우스	7500(
2	네이트	...	7	...	1	프리미엄 게이밍 마우스	7500(

- 고객 테이블과 주문 테이블을 그리고 상품 테이블을 조인을 사용해서 연결했다.

이렇게 `JOIN` 구문은 체인처럼 계속 이어서 작성할 수 있다.

3단계: `WHERE` 절로 조건 필터링하기

이제 두 가지 필터링 조건을 `WHERE` 절에 추가한다. 여러 조건은 `AND` 로 연결한다.

- **거주지 조건:** 주소가 '서울'로 시작하는 모든 고객을 찾기 위해 `LIKE '서울%'` 을 사용했다.
- **날짜 조건:** 6월 날짜의 범위를 지정했다.

```
SELECT
    u.user_id, u.name, u.email, u.address, -- 고객
    o.order_id, o.order_date, o.quantity, -- 주문
    p.product_id, p.name, p.price -- 상품
FROM orders o
JOIN users u ON o.user_id = u.user_id
```

```
JOIN products p ON o.product_id = p.product_id
WHERE u.address LIKE '서울%'
  AND o.order_date >= '2025-06-01' AND o.order_date < '2025-07-01'
```

user_id	name	...	address	order_id	order_date	...	product_id
1	션	...	서울시 강남구	1	2025-06-10 10:00:00	...	1
1	션	...	서울시 강남구	2	2025-06-10 10:05:00	...	4
3	세종대왕	...	서울시 종로구	4	2025-06-12 09:00:00	...	4
5	마리 퀴리	...	서울시 강남구	6	2025-06-16 18:00:00	...	1

4단계: 최종 보고서 양식에 맞게 컬럼 선택 및 정렬하기

마지막으로 문제에서 요구한 컬럼들만 `SELECT` 절에 명시하고, `ORDER BY` 를 사용해 최신순으로 정렬하면 모든 요구사항이 반영된 최종 쿼리가 완성된다.

```
SELECT
    u.name AS customer_name,
    u.email,
    o.order_date,
    p.name AS product_name,
    p.price,
    o.quantity
FROM
    orders o
JOIN
    users u ON o.user_id = u.user_id
JOIN
    products p ON o.product_id = p.product_id
WHERE
    u.address LIKE '서울%'
```

```
    AND o.order_date >= '2025-06-01' AND o.order_date < '2025-07-01'
ORDER BY
    o.order_date DESC;
```

컬럼 별칭(`AS`)을 사용해서 보고서의 헤더를 `customer_name`, `product_name`처럼 더 명확하게 만들어 주었다.

customer_name	email	order_date	product_name	price	quantit
마리 퀴리	marie@example.com	2025-06-16 18:00:00	프리미엄 게이밍 마우스	75000	1
세종대왕	sejong@example.com	2025-06-12 09:00:00	관계형 데이터베이스 입문	28000	1
션	sean@example.com	2025-06-10 10:05:00	관계형 데이터베이스 입문	28000	2
션	sean@example.com	2025-06-10 10:00:00	프리미엄 게이밍 마우스	75000	1

드디어 우리는 흩어져 있던 세 개의 테이블에서 정보를 모으고, 원하는 조건에 맞게 가공하여 완벽한 비즈니스 보고서를 만들어냈다. 이것이 바로 `JOIN`의 진정한 힘이다. 이제 여러분은 `JOIN`을 사용해서 여러 테이블을 자유자재로 연결할 수 있을 것이다.

하지만 `JOIN`만으로 해결하기 어려운 문제들도 존재한다. 예를 들어, **"우리 쇼핑몰에서 가장 비싼 상품을 주문한 고객은 누구일까?"** 라는 질문에 답하려면 어떻게 해야 할까?
먼저 '가장 비싼 상품의 가격'을 알아내고(1단계), 그 가격으로 상품을 찾은 뒤(2단계), 그 상품을 주문한 고객을 찾아야 한다(3단계).

이처럼 여러 단계의 질문을 하나의 쿼리 안에서 논리적으로 해결할 수 있게 해주는 도구가 바로 다음 섹션에서 배울 **서브쿼리(Subquery)**다. 왜 쿼리 안에 또 다른 쿼리를 넣어야 하는지, 다음 시간에 알아보도록 하자.

문제와 풀이1

문제1: 특정 카테고리의 미판매 상품 찾기

[문제]

`products` 테이블과 `orders` 테이블을 `LEFT JOIN` 하여, '전자기기' 카테고리에 속하지만 단 한 번도 판매되지 않은 상품의 이름과 가격을 조회하는 SQL을 작성해라.

[실행 결과]

name	price
스마트 워치	280000

[정답]

```sql
SELECT
    p.name,
    p.price
FROM
    products p
LEFT JOIN
    orders o ON p.product_id = o.product_id
WHERE
    p.category = '전자기기' AND o.order_id IS NULL;
```

- `products` 를 기준 테이블로 삼아 `LEFT JOIN` 을 수행한다.
- `WHERE` 절을 사용하여 `category` 가 '전자기기'인 상품만 필터링하고, 그중에서 `orders` 테이블과 연결고리가 없는 (`o.order_id IS NULL`) 상품, 즉 판매되지 않은 상품을 찾는다.

문제2: 고객별 주문 횟수 구하기 (주문 없는 고객 포함)

[문제]

모든 고객의 이름과 각 고객이 주문한 총 횟수를 조회하는 SQL을 작성해라. 주문을 한 번도 하지 않은 고객은 주문 횟수가 0으로 표시되어야 한다. 결과는 고객 이름으로 오름차순 정렬해라.

[실행 결과]

name	order_count
네이트	2
레오나르도 다빈치	0
마리 퀴리	1
세종대왕	1
션	2
이순신	1

> **힌트**
> 주문이 없는 고객의 경우 `o.order_id`가 `NULL`이므로 `COUNT` 결과가 0이 된다

[정답]

```
SELECT
    u.name,
    COUNT(o.order_id) AS order_count
FROM
    users u
LEFT JOIN
    orders o ON u.user_id = o.user_id
GROUP BY
    u.user_id, u.name
ORDER BY
    u.name;
```

- 모든 고객을 결과에 포함해야 하므로 `users` 테이블을 기준으로 `LEFT JOIN`을 사용한다.

- `COUNT(o.order_id)`를 사용하면, 주문이 없는 고객의 경우 `o.order_id`가 `NULL`이므로 `COUNT` 결과가 0이 된다.
- `GROUP BY`를 사용하여 고객별로 주문 횟수를 집계한다. `u.user_id`를 기준으로 그룹화하는 것이 더 정확하지만, 여기서는 이름도 함께 그룹화하여 `SELECT` 절에 이름을 표시할 수 있도록 했다.

문제3: RIGHT JOIN으로 주문 없는 고객 찾기

[문제]

`RIGHT JOIN`을 사용하여, 가입은 했지만 주문 기록이 없는 고객의 이름과 이메일을 찾는 SQL을 작성해라.

[실행 결과]

name	email
레오나르도 다빈치	vinci@example.com

[정답]

```
SELECT
    u.name,
    u.email
FROM
    orders o
RIGHT JOIN
    users u ON o.user_id = u.user_id
WHERE
    o.order_id IS NULL;
```

- `RIGHT JOIN`을 사용하여 `users` 테이블을 기준 테이블로 만든다. `RIGHT JOIN` 구문에서 오른쪽에 `users` 테이블을 위치시킨다.
- 이렇게 하면 `users` 테이블의 모든 데이터가 결과에 포함되고, 주문 기록이 없는 고객의 `orders` 관련 컬럼은 `NULL`이 된다.
- `WHERE o.order_id IS NULL` 조건으로 주문 기록이 없는 고객만 필터링한다.

문제4: 고객별 주문 상품 목록 조회하기

[문제]

모든 고객의 이름과 그 고객이 주문한 상품의 이름을 조회하는 SQL을 작성해라. 한 고객이 여러 상품을 주문했다면 모든 상품명이 나와야 하며, 주문 기록이 없는 고객의 상품명은 `NULL` 로 표시되어야 한다. 결과는 고객 이름(`user_name`), 상품 이름 순(`product_name`)으로 정렬해라.

[실행 결과]

user_name	product_name
네이트	기계식 키보드
네이트	프리미엄 게이밍 마우스
레오나르도 다빈치	NULL
마리 퀴리	프리미엄 게이밍 마우스
세종대왕	관계형 데이터베이스 입문
션	관계형 데이터베이스 입문
션	프리미엄 게이밍 마우스
이순신	4K UHD 모니터

[정답]

```
SELECT
    u.name AS user_name,
    p.name AS product_name
FROM
    users u
LEFT JOIN
    orders o ON u.user_id = o.user_id
LEFT JOIN
    products p ON o.product_id = p.product_id
ORDER BY
```

```
    user_name, product_name;
```

- `users` 테이블을 기준으로 `orders` 테이블에 `LEFT JOIN`하여 모든 고객을 포함시킨다.
- 그 결과에 다시 한번 `products` 테이블을 `LEFT JOIN`하여 주문된 상품의 이름을 가져온다.
- 연속으로 `LEFT JOIN`을 사용함으로써, `users`(고객)를 최종 기준으로 삼고, 중간에 주문(`orders`)이 없더라도 고객 정보는 유지되며 `product_name`은 `NULL`로 표시된다.

문제와 풀이2

문제1: 특정 상사의 부하 직원 찾기

[문제]

`employees` 테이블을 셀프 조인(SELF JOIN)하여 '최과장'의 직속 부하 직원들의 이름과 직원 ID를 모두 조회하는 SQL 쿼리를 작성해라.

[실행 결과]

employee_id	name	manager_id	manager_name
5	정대리	4	최과장
6	홍사원	4	최과장

[정답]

```sql
SELECT
    e2.employee_id,
    e2.name,
    e2.manager_id,
    e1.name AS manager_name
FROM
    employees e1
JOIN
```

```
    employees e2 ON e1.employee_id = e2.manager_id
WHERE
    e1.name = '최과장';
```

- e1은 상사(manager) 테이블 별칭, e2는 직원(employee) 테이블 별칭이다.

문제2: 모든 상품 옵션 조합에 재질 추가하기

[문제]

`sizes`와 `colors` 테이블을 `CROSS JOIN`하여 모든 색상과 사이즈 조합을 만들었던 것을 응용해 보자. '면(Cotton)'과 '실크(Silk)'라는 두 가지 재질 옵션을 가진 `materials` 테이블을 새로 만들고, 기존의 `sizes`, `colors` 테이블과 모두 `CROSS JOIN`하여 '상품명-색상-사이즈-재질' 형태의 모든 조합을 조회하는 쿼리를 작성해라.

1. `materials` 테이블을 생성하고 데이터를 삽입한다.
2. `sizes`, `colors`, `materials` 세 테이블을 `CROSS JOIN` 한다.
3. `CONCAT` 함수를 사용하여 `상품명-색상-사이즈-재질` 형식의 `product_full_name`을 생성한다.

[실행 결과]

product_full_name	size	color	material
기본티셔츠-Black-L-Cotton	L	Black	Cotton
기본티셔츠-Black-L-Silk	L	Black	Silk
기본티셔츠-Blue-L-Cotton	L	Blue	Cotton
기본티셔츠-Blue-L-Silk	L	Blue	Silk
...	...	...	...
기본티셔츠-Black-XL-Cotton	XL	Black	Cotton
기본티셔츠-Black-XL-Silk	XL	Black	Silk

- 총 24개 행이 출력된다.

[정답]

```sql
-- 실습을 위한 임시 테이블 생성 및 데이터 삽입
-- (테이블이 이미 존재할 경우를 대비하여 DROP 구문 추가)
DROP TABLE IF EXISTS materials;
CREATE TABLE materials (
    material VARCHAR(20) PRIMARY KEY
);
INSERT INTO materials(material) VALUES ('Cotton'), ('Silk');

-- 세 테이블을 CROSS JOIN하여 모든 조합 조회
SELECT
    CONCAT('기본티셔츠-', c.color, '-', s.size, '-', m.material) AS
product_full_name,
    s.size,
    c.color,
    m.material
FROM
    sizes s
CROSS JOIN
    colors c
CROSS JOIN
    materials m
ORDER BY
    s.size, c.color, m.material;
```

문제3: 특정 고객의 주문 내역 상세 조회하기

[문제]

`users`, `orders`, `products` 세 개의 테이블을 조인하여 '네이트' 고객이 주문한 모든 상품의 이름(`product_name`), 주문 날짜(`order_date`), 주문 수량(`quantity`)을 조회하는 SQL 쿼리를 작성해라. 결과는 주문 날짜 최신순으로 정렬해라.

[실행 결과]

customer_name	product_name	order_date	quantity
네이트	프리미엄 게이밍 마우스	2025-06-17 12:00:00	2

네이트	기계식 키보드	2025-06-11 14:20:00	1

[정답]

```
SELECT
    u.name AS customer_name,
    p.name AS product_name,
    o.order_date,
    o.quantity
FROM
    users u
JOIN
    orders o ON u.user_id = o.user_id
JOIN
    products p ON o.product_id = p.product_id
WHERE
    u.name = '네이트'
ORDER BY
    o.order_date DESC;
```

문제4: 서울 지역 고객의 총 주문 금액 계산하기

[문제]

`JOIN`과 집계 함수를 함께 사용하는 종합 문제다.

`users`, `orders`, `products` 테이블을 조인하여 '서울'에 거주하는 고객별로 총 주문 금액(`total_spent`)을 계산하는 쿼리를 작성해라.

총 주문 금액은 각 주문의 `가격(price) * 수량(quantity)`의 합계이다. 결과는 총 주문 금액이 높은 순으로 정렬해라.

[실행 결과]

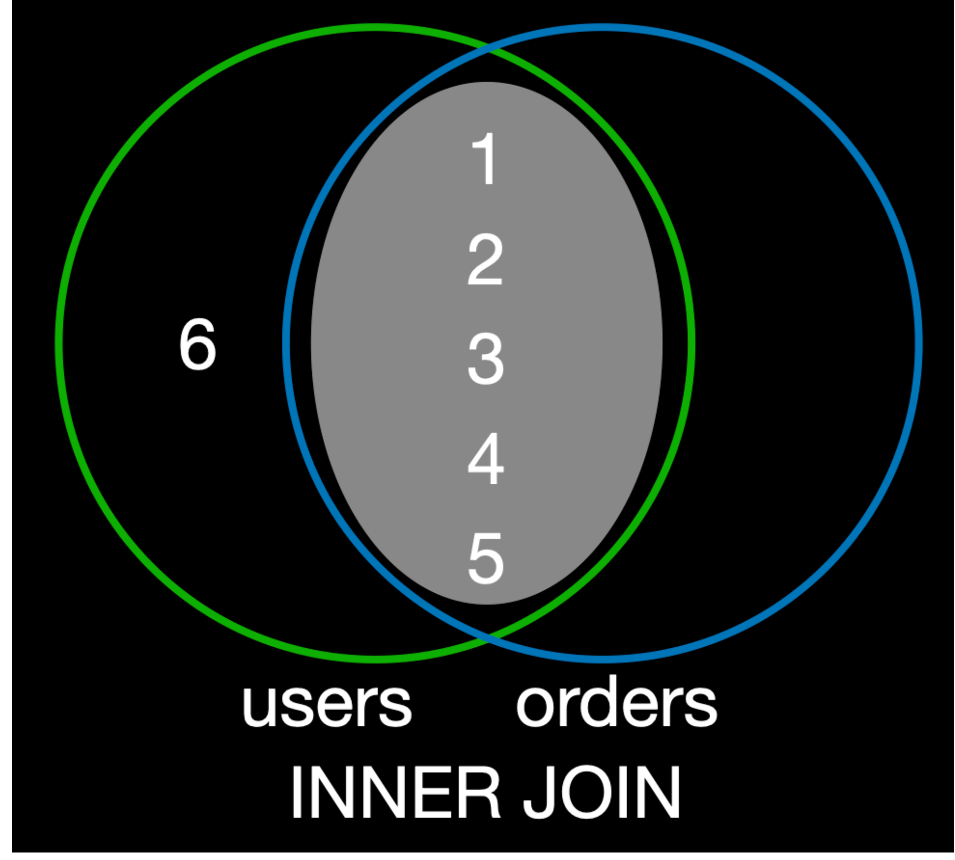
customer_name	total_spent

션	131000
마리 퀴리	75000
세종대왕	28000

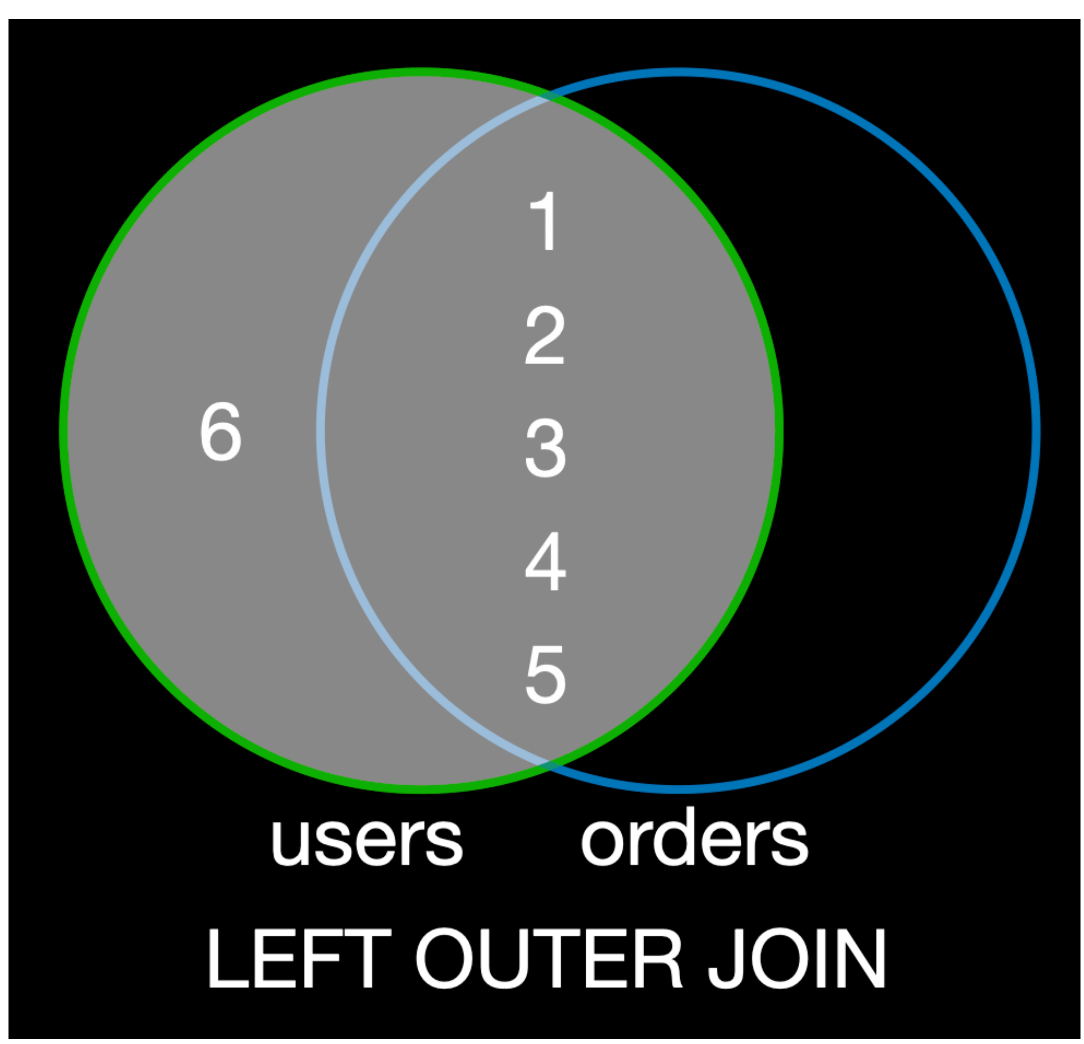
[정답]

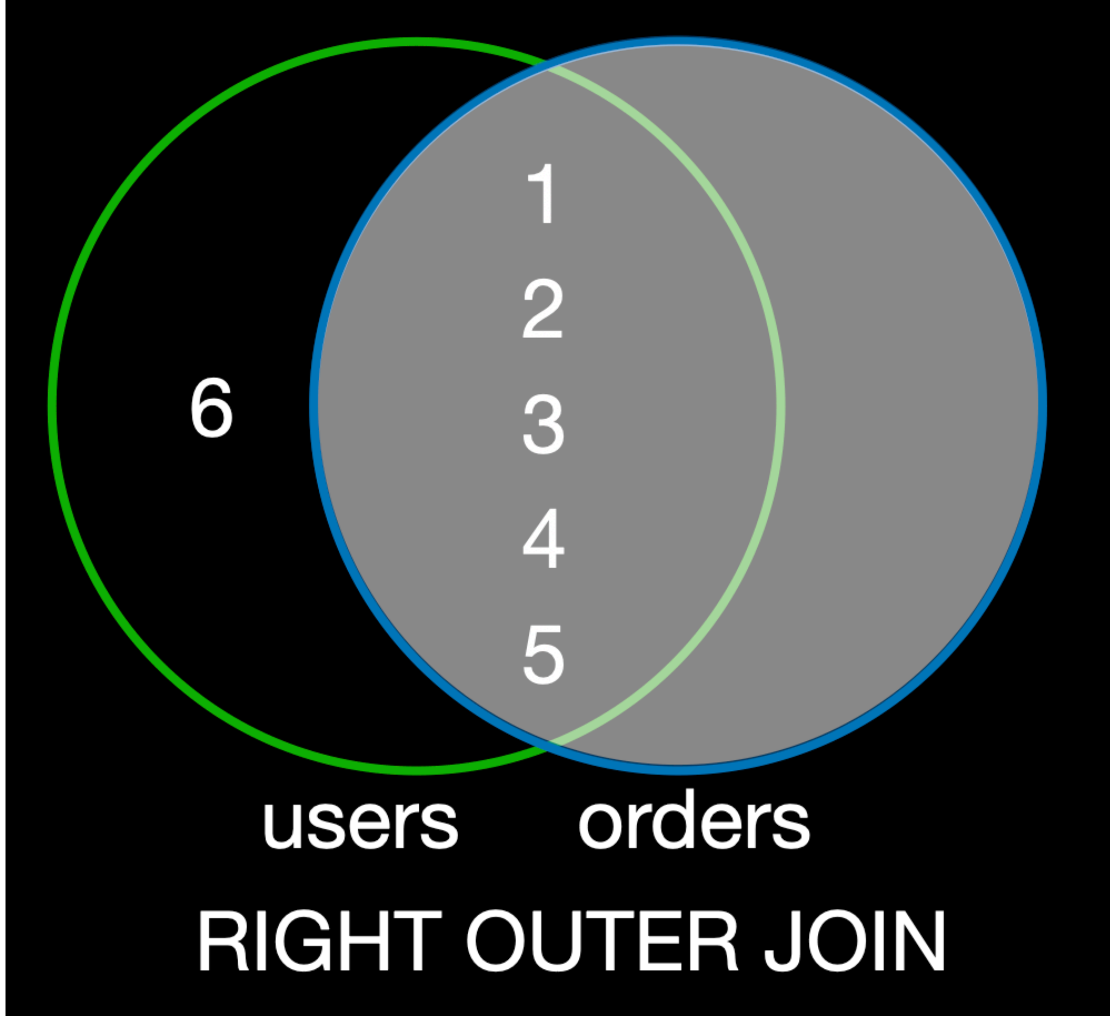
```
SELECT
    u.name AS customer_name,
    SUM(p.price * o.quantity) AS total_spent
FROM
    users u
JOIN
    orders o ON u.user_id = o.user_id
JOIN
    products p ON o.product_id = p.product_id
WHERE
    u.address LIKE '서울%'
GROUP BY
    u.name
ORDER BY
    total_spent DESC;
```

정리

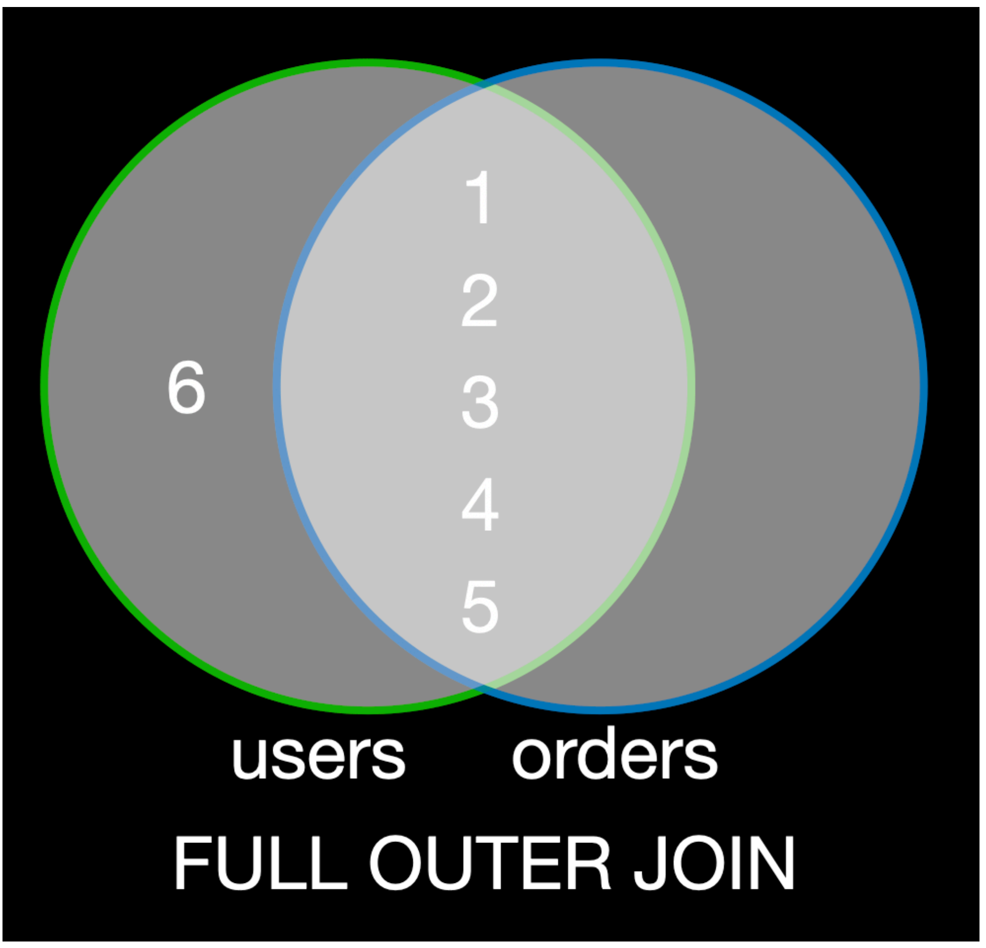


- 내부 조인


1
2
3
4
5
6
users
orders
LEFT OUTER JOIN



- 외부 조인



- 풀 외부 조인 (MySQL 지원 X)

외부 조인 1

- 내부 조인은 양쪽 테이블에 모두 데이터가 존재하는 교집합 영역만 조회한다.
- 한쪽 테이블에만 데이터가 존재하는 경우까지 포함해서 조회하려면 외부 조인을 사용해야 한다.
- 외부 조인은 기준 테이블의 모든 데이터를 결과에 포함시킨다.
- `LEFT JOIN`은 왼쪽 테이블을 기준으로 하고 `RIGHT JOIN`은 오른쪽 테이블을 기준으로 한다.
- 기준 테이블과 짝이 맞는 데이터가 없는 경우 해당 필드는 `NULL` 값으로 채워진다.
- `NULL` 값을 비교할 때는 반드시 `IS NULL` 또는 `IS NOT NULL`을 사용해야 한다.

외부 조인 2

- `LEFT JOIN`과 `RIGHT JOIN`은 테이블의 위치를 바꾸면 서로 동일한 결과를 만들 수 있다.
- `FROM A LEFT JOIN B`는 `FROM B RIGHT JOIN A`와 같은 결과를 반환한다.
- 실무에서는 분석 기준이 되는 테이블을 `FROM` 절에 먼저 쓰고 `LEFT JOIN`으로 다른 테이블을 붙여나가는 방식이 더 직관적이므로 `LEFT JOIN`이 주로 사용된다.

조인의 특징

- 조인 시 행의 개수가 변하는지 여부는 테이블 간의 관계(PK-FK)와 조인 방향에 따라 결정된다.

- 자식 테이블에서 부모 테이블로 조인(FK ➜ PK)하면 자식 테이블의 각 행은 부모의 단 하나의 행과 연결되므로 행 개수가 유지된다.
- 부모 테이블에서 자식 테이블로 조인(PK ➜ FK)하면 부모 테이블의 한 행이 자식의 여러 행과 연결될 수 있으므로 행 개수가 증가할 수 있다.
- 조인으로 인한 행 수 변화를 이해해야 `SUM`, `COUNT` 같은 집계 함수를 정확하게 사용할 수 있다.

셀프 조인

- 셀프 조인은 하나의 테이블을 자기 자신과 조인하는 기법을 말한다.
- 한 테이블 내에 계층적인 관계나 자기 참조 관계가 있을 때 사용한다(예: 직원의 상사 정보).
- 테이블 별칭(Alias)을 사용해 하나의 테이블을 두 개의 다른 테이블처럼 보이게 만드는 것이 핵심이다.
- 상사가 없는 최상위 직원을 포함하려면 `INNER JOIN` 대신 `LEFT JOIN`을 사용한다.

CROSS 조인

- `CROSS JOIN`은 `ON` 조건 없이 한 테이블의 모든 행을 다른 테이블의 모든 행과 각각 연결한다.
- 두 테이블의 모든 경우의 수를 조합한 카테시안 곱(Cartesian Product) 결과를 만든다.
- 상품의 사이즈 색상 재질 등 모든 옵션 조합을 생성할 때 유용하다.
- `INSERT INTO ... SELECT` 구문과 함께 사용하면 마스터 데이터를 쉽게 생성할 수 있다.
- 데이터가 많은 테이블에 사용하면 결과 행이 기하급수적으로 늘어나 서버에 심각한 부하를 줄 수 있으므로 주의해야 한다.

조인 종합 실습

- 실무 문제는 여러 테이블을 동시에 조인하고 여러 조건을 적용해야 해결할 수 있다.
- 문제를 작은 단위로 나누어 필요한 정보 테이블 조건과 정렬 순서를 파악하는 전략이 유용하다.
- `JOIN` 구문은 체인처럼 계속 이어서 여러 테이블을 연결할 수 있다.
- 여러 필터링 조건은 `WHERE` 절에 `AND`로 연결하고 최종 결과는 `ORDER BY`로 정렬한다.