4. 서브쿼리

#0.강의/2.데이터베이스로드맵/2.기본

서브쿼리 소개

조인을 통해 우리는 흩어진 테이블을 연결하는 법을 배웠다. 하지만 데이터에 질문을 던지다 보면, `JOIN`만으로는 한 번에 답하기 어려운, 여러 단계의 사고를 거쳐야 하는 문제들을 만나게 된다.

오늘의 문제 상황을 보자.
"우리 쇼핑몰에서 판매하는 상품들의 평균 가격보다 비싼 상품은 무엇이 있을까?"

이 질문에 답하기 위해 당신은 어떻게 하겠는가? 아마 대부분의 사람들은 자연스럽게 두 단계로 나누어 생각할 것이다.

1단계: 먼저, 전체 상품의 평균 가격을 구한다.
`AVG()` 집계 함수를 사용하면 간단히 구할 수 있다.

```
SELECT AVG(price) FROM products;
```

[실행 결과]

AVG(price)

```
167166.6667
```

평균 가격이 약 167,167.67원이라는 것을 알 수 있다.

2단계: 그 평균 가격보다 비싼 상품을 찾는다.
이제 위에서 구한 값을 `WHERE` 절에 직접 넣어서 쿼리를 실행한다.

```sql
SELECT name, price
FROM products
WHERE price > 167166.67;
```

[실행 결과]

name	price
4K UHD 모니터	350000
스마트 워치	280000

이렇게 두 번의 쿼리를 통해 우리는 원하는 결과를 얻을 수 있었다.

아마도 이런 생각이 들 것이다. "이 과정을 하나의 쿼리로 합칠 수는 없을까?"

두 번에 걸쳐 쿼리를 실행하는 방식은 몇 가지 문제점을 가진다.

- **번거롭다:** 매번 첫 번째 쿼리의 결과를 복사해서 두 번째 쿼리에 붙여 넣어야 한다.
- **오류에 취약하다:** 만약 상품 데이터가 실시간으로 추가되거나 가격이 변경된다면 어떨까? 1단계 쿼리를 실행하고 2단계 쿼리를 실행하는 그 짧은 순간에도 평균 가격은 변할 수 있다. 이렇게 되면 잘못된 기준으로 데이터를 조회하게 될 수도 있다.

우리는 이 두 단계를 논리적으로 완벽한 **하나의 작업 단위**로 묶고 싶다. 이럴 때 사용하는 기술이 바로 **서브쿼리(Subquery)**이다.

서브쿼리의 개념

서브쿼리는 말 그대로 **하나의 SQL 쿼리 문 안에 포함된 또 다른** `SELECT` **쿼리**를 의미한다. 바깥쪽의 메인쿼리가 실행

되기 전에, 괄호 `()` 안에 있는 서브쿼리가 먼저 실행된다. 그리고 데이터베이스는 서브쿼리의 실행 결과를 바깥쪽 메인 쿼리에게 전달하여, 메인쿼리가 그 결과를 사용해서 최종 작업을 수행하게 된다.

방금 우리가 두 단계로 풀었던 문제를 서브쿼리를 사용하면 이렇게 하나의 쿼리로 표현할 수 있다.

```
SELECT name, price
FROM products
WHERE price > (SELECT AVG(price) FROM products);
```

해당 쿼리의 실행 순서는 다음과 같다.

1. 데이터베이스는 괄호 안의 서브쿼리, `SELECT AVG(price) FROM products`를 **가장 먼저** 실행한다.
2. 서브쿼리가 실행된 결과인 `167166.67`이라는 단일 값을 얻는다.
3. 이제 원래의 쿼리는 내부적으로 다음과 같이 변한다.
   `SELECT name, price FROM products WHERE price > 167166.67;`
4. 최종적으로 이 변환된 메인쿼리가 실행되어 우리에게 결과를 보여준다.

결과는 우리가 두 단계로 나누어 실행했을 때와 완벽하게 동일하지만, 이제 단 하나의 쿼리로 해결했다.

서브쿼리 종류와 특징

서브쿼리는 방금 본 `WHERE` 절 외에도 쿼리의 다양한 위치에서 활약하며 각기 다른 역할을 수행한다.
서브쿼리는 반환하는 **행과 컬럼의 수**에 따라 종류가 나뉘며, 사용되는 **위치와 연산자**에 따라 그 역할이 결정된다.

구분	반환 형태	주요 사용 위치	연산자 / 구문	명칭	핵심 용도
단일 컬럼	단일 행	`SELECT`, `WHERE`, `HAVING`	`=`, `>`, `<` 등	- **스칼라 서브쿼리** (Scalar Subquery) - 단일 값 서브쿼리	**단일 값**이 필요한 모든 곳 (값 비교, 특정 값 표시)

	다중 행	`WHERE`, `HAVING`	`IN`, `ANY`, `ALL`	- **다중 행 서브쿼리** - 값 목록 서브쿼리 - 리스트 서브쿼리 - 컬럼 서브쿼리	**값 목록(List)**과 비교
다중 컬럼	단일 행	`WHERE`, `HAVING`	`(c1, c2) = ...`	- **다중 컬럼 서브쿼리** - 행 서브쿼리	여러 컬럼 값을 **정확히 1:1로 비교** (쌍 비교)
	다중 행	`WHERE`, `HAVING`	`(c1, c2) IN ...`	- **다중 컬럼 서브쿼리**	여러 컬럼 **조합**이 목록에 포함되는지 비교 (튜플 비교)
다중 컬럼	다중 행	`FROM`	`FROM (...) AS alias`	- 테이블 서브쿼리 - 인라인 뷰 - 파생 테이블	**가상의 테이블**을 생성하여 다른 테이블과 `JOIN` 등 재가공

- 테이블 서브쿼리는 표에는 다중 컬럼 다중 행으로 표기했지만 컬럼, 행 수가 단일이어도 쓸 수 있다. 실무에선 통상 다중 컬럼, 다중 행에 자주 사용한다.
- 각 종류별로 참으로 다양한 명칭이 사용된다. 외우지는 말고 예제를 진행하면서 자연스럽게 이해하자.

서브쿼리는 `JOIN`과 함께 복잡한 데이터를 분석하는 양대 산맥과도 같은 기술이다. `JOIN`이 테이블을 수평으로 넓혀나가는 기술이라면, 서브쿼리는 쿼리 내부에 논리적인 단계를 만들어 안으로 깊게 파고드는 기술이라고 할 수 있다.

지금부터 이 표에 정리한 내용을 하나씩 알아가보자.

스칼라 서브쿼리

지난 시간, 우리는 여러 단계의 계산을 하나의 쿼리로 합쳐주는 서브쿼리의 필요성과 기본 개념에 대해 알아보았다. 이제부터는 서브쿼리가 사용되는 위치와 반환하는 결과의 형태에 따라 어떻게 나뉘고 활용되는지 본격적으로 알아보자.

가장 먼저 배울 것은 **단일 컬럼, 단일 행 서브쿼리**다. 이름에서 알 수 있듯이, 서브쿼리를 실행했을 때 그 결과가 **오직 하나의 행, 하나의 컬럼**으로 나오는 경우를 말한다.

서브쿼리의 결과가 하나의 값으로 정해지기 때문에 이것을 스칼라 서브쿼리라 한다.

> 용어 - 스칼라
> **스칼라(Scalar)**는 원래 수학과 물리학에서 온 단어로, **'단 하나의 값'**을 의미한다.

결과가 '하나의 값'으로 정해지기 때문에, 우리는 이 값을 익숙한 단일 행 비교 연산자들(`=`, `>`, `<`, `>=`, `<=`, `<>`)과 함께 사용할 수 있다. "A는 B보다 크다"처럼, 비교 대상 B가 하나의 명확한 값이어야 말이 되는 것과 같은 이치다.

오늘의 문제 상황을 통해 단일 행 서브쿼리를 이해해 보자.

"특정 주문(order_id = 1)을 한 고객과 같은 도시에 사는 모든 고객을 찾고 싶다."

서브쿼리 없이 2단계로 해결하기

먼저 서브쿼리 없이 이 문제를 해결하는 과정을 따라가 보자.

1단계: `order_id`가 1인 고객의 도시 알아내기
`order_id`가 1인 주문을 찾고, 그 주문을 한 고객의 `user_id`를 알아낸 뒤, `users` 테이블과 조인하여 그 고객의 주소(도시)를 알아내야 한다.

```sql
SELECT u.address
FROM users u
JOIN orders o ON u.user_id = o.user_id
WHERE o.order_id = 1;
```

- `orders` 테이블을 보면, `order_id` 1번은 `user_id` 1번('션')이 주문했다. `users` 테이블에서 '션'의 주소는 '서울시 강남구'다.

[실행 결과]

address
서울시 강남구

주문자 '션'이 '서울시 강남구'에 산다는 것을 알아냈다.

2단계: 해당 도시에 사는 모든 고객 찾기

이제 위에서 얻은 '서울시 강남구'라는 값을 이용해 `users` 테이블에서 동일한 주소를 가진 고객을 모두 찾는다.

```
SELECT name, address
FROM users
WHERE address = '서울시 강남구';
```

[실행 결과]

name	address
션	서울시 강남구
마리 퀴리	서울시 강남구

결과를 보면 주문을 한 '션'뿐만 아니라, 같은 '서울시 강남구'에 사는 '마리 퀴리'도 함께 조회되는 것을 확인할 수 있다.
이렇게 우리는 2단계에 걸쳐 원하는 결과를 얻었다.

단일 행 서브쿼리로 해결하기

이제 이 두 단계를 하나의 쿼리로 합쳐보자. 1단계 쿼리가 "비교할 기준값(도시 이름)을 찾아내는" 역할을 하므로, 이 쿼리를 통째로 서브쿼리로 만들어 `WHERE` 절의 비교 대상 위치에 넣어주면 된다.

```
SELECT name, address
FROM users
WHERE address = (SELECT u.address
                 FROM users u
                 INNER JOIN orders o ON u.user_id = o.user_id
                 WHERE o.order_id = 1);
```

[실행 결과]

name	address
션	서울시 강남구
마리 퀴리	서울시 강남구

쿼리 실행 흐름

1. 괄호 안의 서브쿼리가 먼저 실행되어 단일 값 `'서울시 강남구'`를 반환한다.
2. 메인쿼리는 `WHERE address = '서울시 강남구'` 와 동일한 형태로 바뀐다.
3. 최종적으로 메인쿼리가 실행되어 우리에게 원하는 결과를 보여준다. 결과는 당연히 2단계로 나누어 실행했을 때와 동일하다.

이처럼 서브쿼리를 사용하면 여러 번의 쿼리를 하나로 합쳐 코드를 간결하게 만들고, 애플리케이션과 데이터베이스 간의 통신 횟수를 줄여 성능상 이점을 얻을 수 있다.

스칼라 서브쿼리의 치명적 오류

단일 행을 반환하는 스칼라 서브쿼리를 사용할 때 가장 주의해야 할 점은, 서브쿼리의 결과가 **반드시, 무슨 일이 있어도, 단 하나의 행만 반환해야 한다**는 것이다. 만약 서브쿼리가 두 개 이상의 행을 반환하면 어떻게 될까?

데이터베이스는 `address = ('결과1', '결과2')` 와 같은 형태의 비교를 이해할 수 없다. `address`가 '결과1'과 같다는 것인가? 아니면 '결과2'와 같다는 것인가? 하나의 값과 비교해야 하는 `=` 연산자는 비교 대상이 여러 개 들어오면 무엇을 해야 할지 알 수 없게 된다.

예를 들어, "서울 또는 성남에 사는 고객의 주소를 모두 조회"하는 서브쿼리를 만들어 비교한다고 가정해 보자.

```
-- 이 쿼리는 의도적으로 오류를 발생시킨다.
SELECT name, address
FROM users
WHERE address = (SELECT address FROM users WHERE name IN ('션', '네이트'));
```

괄호 안의 서브쿼리 `SELECT address FROM users WHERE name IN ('션', '네이트')`는 '서울시 강남구'와 '경기도 성남시'라는 **두 개의 행**을 반환한다.

메인쿼리의 `WHERE address = ...` 비교문은 이 두 개의 값을 어떻게 처리해야 할지 몰라 혼란에 빠진다. 결국, 데이터베이스는 이런 에러를 발생시키며 실행을 멈춘다.

```
Error Code: 1242. Subquery returns more than 1 row
```

이처럼 단일 행 비교 연산자(`=`, `>`, `<` 등)를 사용할 때는, 서브쿼리의 결과가 반드시 **단일 행**일때만 사용해야 한다. 예를 들어 `order_id`나 `user_id`처럼 `PK`나 `UNIQUE` 제약 조건이 걸린 컬럼을 조건으로 조회하는 경우가 대표적이다.

그렇다면 서브쿼리가 여러 개의 행을 반환하는 것이 당연한 상황에서는 어떻게 해야 할까? 예를 들어, "전자기기 카테고리에 속하는 모든 상품들을 주문한 고객 목록을 찾아라" 와 같은 문제 말이다. 이때는 `=` 같은 단일 행 연산자가 아닌, 여러 개의 값을 다룰 수 있는 `IN`과 같은 **다중 행 연산자**를 사용해야 한다.

다음 시간에는 이처럼 서브쿼리의 결과가 여러 행일 때 사용하는 **다중 행 서브쿼리**에 대해 알아보겠다.

다중 행 서브쿼리

지난 시간에는 서브쿼리의 결과가 단 하나의 행만 반환하는 '스칼라 서브쿼리'에 대해 배웠다. 그리고 서브쿼리가 두 개 이상의 행을 반환할 때, `=` 같은 단일 행 비교 연산자를 사용하면 에러가 발생한다는 것도 확인했다.

하지만 현실의 문제는 종종 여러 개의 결과를 반환하는 서브쿼리를 필요로 한다. 오늘의 문제 상황을 보자.
"'전자기기' 카테고리에 속한 모든 상품들을 주문한 주문 내역을 전부 보고 싶다."

이 문제를 풀기 위한 우리의 생각의 흐름은 이렇다.

1. 먼저, '전자기기' 카테고리에 속한 상품들의 `product_id`를 모두 찾는다.
2. 그다음, `orders` 테이블에서 `product_id`가 우리가 찾아낸 `product_id` 목록 안에 포함된 주문들을 모두 찾는다.

여기서 1단계의 결과는 당연히 여러 개일 것이다. 우리 쇼핑몰에는 여러 종류의 '전자기기' 상품이 있기 때문이다. 이처럼 **서브쿼리의 결과가 여러 행으로 반환되는 것이 당연할 때 사용하는 것이 바로 다중 행 서브쿼리**다.

다중 행 서브쿼리의 결과를 처리하기 위해서는 `=` 같은 단일 행 연산자가 아닌, 목록을 다룰 수 있는 특별한 연산자가 필요하다. 대표적으로 `IN`, `ANY`, `ALL`이 있다.

`IN` 연산자: 목록에 포함된 값과 일치하는지 확인

`IN` 연산자는 다중 행 서브쿼리와 함께 가장 흔하게 사용되는, 가장 직관적인 연산자다. `WHERE 컬럼명 IN (값1, 값2, ...)` 처럼, 특정 컬럼의 값이 괄호 안의 목록 중 하나라도 일치하면 참(true)을 반환한다.

자, 이제 `IN`과 서브쿼리를 조합하여 위의 문제를 한 번에 해결해 보자.

1단계 (서브쿼리): '전자기기' 상품의 ID 목록 조회

먼저 서브쿼리가 될 부분부터 확인해 보자.

```
SELECT product_id FROM products WHERE category = '전자기기'
ORDER BY product_id;
```

- 확인의 편의상 정렬을 추가했다. 서브쿼리에 사용할 때는 정렬을 빼는 것이 좋다.

[실행 결과]

product_id
1
2
3
6

예상대로 '전자기기' 카테고리 상품들의 `id` 목록 `(1, 2, 3, 6)`이 반환되었다.

2단계 (메인쿼리): `IN`을 이용해 최종 결과 조회

이제 이 쿼리를 서브쿼리로 사용하여, `orders` 테이블에서 `product_id`가 위 목록에 포함된 주문들만 필터링한다.

```
SELECT * FROM orders
WHERE product_id IN (SELECT product_id
                     FROM products
                     WHERE category = '전자기기')
ORDER BY order_id;
```

쿼리 실행 흐름

1. 괄호 안의 서브쿼리가 먼저 실행되어 `(1, 2, 3, 6)` 이라는 `id` 목록을 반환한다.
2. 메인쿼리는 `WHERE product_id IN (1, 2, 3, 6)` 과 동일한 형태로 바뀐다. 즉, `product_id`가 1이거나, 2거나, 3이거나, 6인 모든 주문을 찾아낸다.
3. 최종적으로 메인쿼리가 실행된다.

쉽게 이야기하면 서브쿼리가 `(1, 2, 3, 6)`의 값 목록으로 변환되어서 다음 쿼리가 실행되는 것이다.

```
SELECT * FROM orders
WHERE product_id IN (1,2,3,6)
ORDER BY order_id;
```

[실행 결과]

order_id	user_id	product_id	order_date	quantity	status
1	1	1	2025-06-10 10:00:00	1	COMPLETED
3	2	2	2025-06-11 14:20:00	1	SHIPPED
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

이렇게 `IN` 연산자를 활용하여 여러 결과를 반환하는 서브쿼리를 깔끔하게 처리했다. 참고로 목록에 없는 것을 찾고 싶을 때는 `NOT IN`을 사용하면 된다.

`ANY`, `ALL` 연산자: 목록의 모든/일부 값과 비교

`ANY`와 `ALL`은 주로 `>`, `<`와 같은 비교 연산자와 함께 사용되어, 서브쿼리가 반환한 여러 값들과 비교하는 역할을 한다.

- `> ANY (서브쿼리)` : 서브쿼리가 반환한 여러 결과값 중 **어느 하나보다만 크면** 참이다. 즉, **최소값보다 크면** 참이 된다.
- `> ALL (서브쿼리)` : 서브쿼리가 반환한 여러 결과값 **모두보다 커야만** 참이다. 즉, **최대값보다 커야** 참이 된다.
- `< ANY (서브쿼리)` : 최대값보다 작으면 참이다.

- `< ALL (서브쿼리)` : 최소값보다 작으면 참이다.
- `= ANY (서브쿼리)` : `IN`과 완전히 동일한 의미다. 목록 중 어느 하나와 같으면 참이다.

개념이 조금 헷갈릴 수 있으니, 간단한 예시로 이해해 보자.
어떤 서브쿼리가 `(100, 200, 300)` 이라는 세 개의 값을 반환했다고 가정하자.

- `WHERE price > ANY (100, 200, 300)`
  - 이 조건은 `price > 100` 과 같다. 100보다 크기만 하면 200이나 300보다 작더라도, 목록 중 100보다는 크기 때문에 참이 된다.
- `WHERE price > ALL (100, 200, 300)`
  - 이 조건은 `price > 300` 과 같다. 목록의 모든 값, 즉 100, 200, 300보다 모두 커야 하므로, 결국 최대값인 300보다 커야 참이 된다.

실제 쇼핑몰 데이터로 `ANY`, `ALL` 사용해보기

우리 쇼핑몰 데이터에 직접 적용해 보면서 개념을 확실히 다져보자.

문제 상황 1: '전자기기' 카테고리의 어떤 상품보다도 비싼 상품 찾기
쇼핑몰의 MD(상품 기획자)가 이런 요청을 했다고 가정해 보자. "'전자기기' 카테고리에 있는 상품들보다 비싼 상품들은 어떤 것들이 있는지 리스트를 뽑아주세요. 기준은 '전자기기' 카테고리의 가장 저렴한 상품보다만 비싸면 됩니다."

이것이 바로 `> ANY`를 사용하기 좋은 상황이다. '전자기기' 카테고리 상품들의 가격 목록 중, 어느 하나보다만 크면 되기 때문이다.

먼저, 서브쿼리를 통해 '전자기기' 카테고리의 상품 가격들을 확인해 보자.

```sql
SELECT price FROM products WHERE category = '전자기기';
```

[실행 결과]

price
75000
120000
350000

280000

서브쿼리는 `(75000, 120000, 350000, 280000)`이라는 가격 목록을 반환한다. 이제 `ANY`를 사용하여 이 목록의 최소값(75000)보다 비싼 모든 상품을 조회해 보자.

```
SELECT name, price
FROM products
WHERE price > ANY (SELECT price FROM products WHERE category = '전자기기');
```

이 쿼리는 `price > 75000` OR `price > 120000` OR `price > 350000` OR `price > 280000` 와 같이 동작한다. 결국 `price`가 목록의 최소값인 75000보다 크기만 하면 이 조건이 참이 되므로, `price > 75000`과 동일한 결과를 낸다.

[실행 결과]

name	price
기계식 키보드	120000
4K UHD 모니터	350000
고급 가죽 지갑	150000
스마트 워치	280000

결과를 보면 '프리미엄 게이밍 마우스'(75,000원)를 제외하고, 그보다 비싼 모든 상품이 조회된 것을 확인할 수 있다. 심지어 같은 '전자기기' 카테고리에 속한 '기계식 키보드', '4K UHD 모니터', '스마트 워치'도 포함된다.

문제 상황 2: '전자기기' 카테고리의 모든 상품보다 비싼 상품 찾기
이번에는 MD가 좀 더 까다로운 요청을 했다. "전자기기 카테고리의 그 어떤 상품보다도 비싼, 즉 우리 쇼핑몰 최고가 전자기기 제품보다 더 비싼 상품이 있는지 궁금합니다. 그런 '프리미엄' 상품 리스트를 뽑아주세요."

이것이 바로 `> ALL`을 사용할 완벽한 시나리오다. '전자기기' 카테고리 상품들의 모든 가격보다 커야 하기 때문이다.

```sql
SELECT name, price
FROM products
WHERE price > ALL (SELECT price FROM products WHERE category = '전자기기');
```

서브쿼리는 이전과 동일하게 `(75000, 120000, 350000, 280000)`를 반환한다. `> ALL` 조건은 이 모든 값보다 커야 하므로, 목록의 최대값인 350000보다 커야 참이 된다. 즉, `price > 350000`과 동일하게 동작한다.

[실행 결과]
(결과 없음)

현재 우리 `products` 테이블에는 350,000원보다 비싼 상품이 없으므로 아무런 결과도 반환되지 않는다.

실무 팁: `MIN`, `MAX` 집계 함수와의 비교
`ANY`와 `ALL`은 특정 조건(예: 특정 부서 직원들의 모든 급여보다 많은 급여를 받는 사람 찾기)을 해결하는 데 유용하지만, 사실 실무에서는 `IN` 연산자나 `MIN()`, `MAX()` 같은 집계 함수를 이용한 서브쿼리로 대체할 수 있는 경우가 많다. 오히려 집계 함수를 쓰는 것이 코드가 더 명확하고 직관적일 때가 많다.

예를 들어, 앞서 작성한 `ANY`, `ALL` 쿼리는 다음과 같이 바꿀 수 있다.

`> ANY` 쿼리 대체 (`MIN` 사용)

```sql
SELECT name, price
FROM products
WHERE price > (SELECT MIN(price) FROM products WHERE category = '전자기기');
```

[실행 결과]

name	price
기계식 키보드	120000
4K UHD 모니터	350000
고급 가죽 지갑	150000
스마트 워치	280000

- 앞서 `ANY`를 사용한 실행 결과와 같다.

`> ALL` 쿼리 대체 (`MAX` 사용)

```
SELECT name, price
FROM products
WHERE price > (SELECT MAX(price) FROM products WHERE category = '전자기기');
```

[실행 결과]

(결과 없음)

- 앞서 `ALL`을 사용한 실행 결과와 같다.

두 쿼리 모두 `ANY`, `ALL`을 사용했을 때와 완전히 동일한 결과를 반환한다. 많은 개발자들이 `ANY`, `ALL` 보다는 `MIN`, `MAX`를 사용한 코드를 더 이해하기 쉽다고 생각한다. 그렇기 때문에 `ANY`, `ALL`의 사용 빈도는 `IN` 이나 집계 함수에 비해 낮은 편이다.

우리가 지금까지 배운 서브쿼리는 결과로 하나의 값(단일 행, 단일 컬럼)을 반환하거나, 값의 목록(다중 행, 단일 컬럼)을 반환하는 경우였다. 그런데 만약 서브쿼리가 **하나의 행**을 반환하지만, 그 안에 **여러 개의 컬럼**을 담고 있다면 어떻게 해야 할까?

예를 들어, "특정 상품과 판매자도 같고, 카테고리도 같은 다른 상품을 찾아라" 와 같은, 여러 조건을 동시에 만족하는 경우를 찾아야 할 때가 있다.

바로 이럴 때 **다중 컬럼 서브쿼리(Multi-Column Subquery)**를 사용한다. 이름 그대로, 서브쿼리가 반환하는 결과에 여러 개의 컬럼이 포함되는 경우를 말한다.

다중 컬럼 서브쿼리

이전까지 우리는 서브쿼리가 하나의 컬럼만을 반환하는 경우만 다루었다. 하지만 서브쿼리는 SELECT 문이므로, 당연히 여러 개의 컬럼을 반환할 수도 있다.

다중 컬럼 서브쿼리(Multi-Column Subquery)는 서브쿼리의 `SELECT` 절에 두 개 이상의 컬럼이 포함되는 경우를 말한다. 이 기법은 메인쿼리의 `WHERE` 절에서 여러 컬럼을 **동시에 비교**해야 할 때 매우 유용하다.

오늘의 문제 상황을 통해 다중 컬럼 서브쿼리가 왜 필요한지 알아보자.

"우리 쇼핑몰의 고객 '네이트'(`user_id`=2)가 한 주문(`order_id`=3)이 있다. 이 주문과 동일한 고객이면서 주문 처리 상태(`status`)도 같은 모든 주문을 찾아보자."

이 문제의 핵심은 `user_id`와 `status`라는 **두 개의 조건**을 동시에 만족하는 주문을 찾는 것이다.

다중 컬럼 서브쿼리로 문제 해결하기

이 문제를 해결하기 위해, 우리는 서브쿼리를 통해 `order_id`가 3인 주문의 `user_id`와 `status`를 한 번에 조회하고, 메인쿼리에서는 이 두 값을 동시에 비교할 것이다.

1단계 (서브쿼리): 비교 기준이 될 고객 ID와 주문 상태 조회

먼저 `order_id`가 3인 주문의 고객 ID와 주문 상태를 조회하는 서브쿼리를 만들어보자.

```
SELECT user_id, status FROM orders WHERE order_id = 3;
```

`order_id`는 `PK`이므로 이 쿼리는 반드시 **하나의 행**만 반환한다.

[실행 결과]

user_id	status
2	SHIPPED

서브쿼리는 `(2, 'SHIPPED')`라는 단일 행, 다중 컬럼 결과를 반환한다.

2단계 (메인쿼리): 다중 컬럼 비교

이제 이 서브쿼리를 메인쿼리의 `WHERE` 절에 넣어서 두 개의 컬럼을 한 번에 비교한다. 이때, 메인쿼리의 `WHERE` 절에도 비교할 컬럼들을 괄호로 묶어 `(user_id, status)`와 같은 형태로 작성해주어야 한다.

```
SELECT order_id, user_id, status, order_date
FROM orders
WHERE (user_id, status) = (SELECT user_id, status
```

```
                            FROM orders
                            WHERE order_id = 3);
```

이 쿼리는 다음과 같은 형태로 변환된다.

```
SELECT order_id, user_id, status, order_date
FROM orders
WHERE (user_id, status) = (2, 'SHIPPED');
```

쿼리 실행 흐름

1. 괄호 안의 서브쿼리가 먼저 실행되어 `(2, 'SHIPPED')` 라는 한 쌍의 값을 반환한다.
2. 메인쿼리는 `WHERE (user_id, status) = (2, 'SHIPPED')` 와 동일한 형태로 바뀐다. 이것은 `WHERE user_id = 2 AND status = 'SHIPPED'` 와 논리적으로 같다.
3. 최종적으로 `orders` 테이블에서 `user_id` 가 2이고 `status` 가 'SHIPPED'인 모든 주문을 찾아 반환한다.

[실행 결과]

order_id	user_id	status	order_date
3	2	SHIPPED	2025-06-11 14:20:00
7	2	SHIPPED	2025-06-17 12:00:00

결과를 보면 기준이 된 주문(`order_id` =3)을 포함하여, 고객('네이트', `id=2`)과 주문 상태('SHIPPED')가 모두 동일한 주문들이 성공적으로 조회된 것을 확인할 수 있다.

만약 여기서 기준이 된 주문 자신을 제외하고 싶다면, `WHERE` 절에 조건을 하나 더 추가하면 된다.

```
SELECT order_id, user_id, status, order_date
FROM orders
WHERE (user_id, status) = (SELECT user_id, status
                           FROM orders
                           WHERE order_id = 3)
  AND order_id != 3; -- 자기 자신은 제외
```

[실행 결과]

order_id	user_id	status	order_date
7	2	SHIPPED	2025-06-17 12:00:00

주의할 점

다중 컬럼 서브쿼리를 `=` 연산자와 함께 사용할 때는, 단일 행 서브쿼리와 마찬가지로 서브쿼리의 결과가 **반드시 하나의 행**이어야 한다는 점을 잊지 말아야 한다. 만약 서브쿼리가 두 개 이상의 행을 반환하면 데이터베이스는 어떤 행과 비교해야 할지 알 수 없으므로 오류를 발생시킨다.

다중 컬럼 서브쿼리와 `IN` 연산자

다중 컬럼 서브쿼리 역시 서브쿼리의 결과가 여러 행일 수 있다. 이때는 `=` 대신 `IN` 연산자를 사용해야 한다.
예를 들어, **"각 고객별로 가장 먼저 한 주문의 주문ID, 사용자ID, 사용자이름, 제품이름, 주문 날짜를 조회해라"** 와 같은 요구사항이 있을 수 있다.

먼저 각 고객별로 가장 빠른 주문 날짜를 구하는 서브쿼리를 작성해보자.

```
SELECT user_id, MIN(order_date)
FROM orders
GROUP BY user_id;
```

[실행 결과]

user_id	MIN(order_date)
1	2025-06-10 10:00:00
2	2025-06-11 14:20:00
3	2025-06-12 09:00:00
4	2025-06-15 11:30:00
5	2025-06-16 18:00:00

이 서브쿼리는 여러 고객의 첫 주문 정보를 담고 있으므로, 여러 행과 여러 열을 반환한다.

이제 이 서브쿼리의 결과를 이용해 각 고객의 첫 주문 정보를 조회하자. 우선 단순하게 조회해보자.
이때는 서브쿼리가 여러 개의 `(user_id, order_date)` 쌍을 반환하므로 `=` 연산자 대신 `IN`을 사용해야 한다.

```
SELECT
    o.order_id,
    o.user_id,
    o.order_date
FROM orders o
WHERE (o.user_id, o.order_date) IN (
    SELECT user_id, MIN(order_date)
    FROM orders
    GROUP BY user_id
);
```

[실행 결과]

order_id	user_id	order_date
1	1	2025-06-10 10:00:00
3	2	2025-06-11 14:20:00
4	3	2025-06-12 09:00:00
5	4	2025-06-15 11:30:00
6	5	2025-06-16 18:00:00

- 각 고객별 첫 주문

이제 필요한 주문ID, 사용자ID, 사용자이름, 제품이름, 주문 날짜를 모두 조회해보자.

```
SELECT
    o.order_id,
    o.user_id,
    u.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.user_id, o.order_date) IN (
    SELECT user_id, MIN(order_date)
    FROM orders
    GROUP BY user_id
);
```

[실행 결과]

order_id	user_id	name	product_name	order_date
1	1	션	프리미엄 게이밍 마우스	2025-06-10 10:00:00
3	2	네이트	기계식 키보드	2025-06-11 14:20:00
4	3	세종대왕	관계형 데이터베이스 입문	2025-06-12 09:00:00
5	4	이순신	4K UHD 모니터	2025-06-15 11:30:00
6	5	마리 퀴리	프리미엄 게이밍 마우스	2025-06-16 18:00:00

메인쿼리의 각 행에 대해 `(user_id, order_date)` 쌍이 서브쿼리가 반환한 여러 쌍 중 하나와 일치하는 경우에만 최종 결과에 포함된다.

이처럼 다중 컬럼 서브쿼리는 두 개 이상의 속성이 **조합**되어 특정 의미를 가질 때, 그 조합 자체를 조건으로 사용하여 데이터를 조회하는 간결하고 강력한 방법을 제공한다.

상관 서브쿼리1

이전 시간까지 우리는 `WHERE` 절에 서브쿼리를 사용하여, 메인쿼리와는 독립적으로 실행된 결과를 필터링 조건으로 사용하는 법을 배웠다. 즉, 서브쿼리가 먼저 한 번 실행되어 값을 만들어내면, 메인쿼리가 그 값을 이어받아 사용하는 방식이었다.

하지만 이런 질문에는 어떻게 답해야 할까?
"각 상품별로, 자신이 속한 카테고리의 평균 가격 이상의 상품들을 찾아라."

이 문제의 핵심은 '전체 평균 가격'이 아닌, **'자신이 속한 바로 그 카테고리의 평균 가격'**과 비교해야 한다는 점이다.

- '프리미엄 게이밍 마우스'는 '전자기기' 카테고리에 속한다. '전자기기' 카테고리의 평균 가격과 비교해야 한다.
- '관계형 데이터베이스 입문' 책은 '도서' 카테고리에 속한다. '도서' 카테고리의 평균 가격과 비교해야 한다.
- '고급 가죽 지갑'은 '패션' 카테고리에 속한다. '패션' 카테고리의 평균 가격과 비교해야 한다.

이처럼 서브쿼리가 메인쿼리에서 현재 처리 중인 행의 특정 값(예: 카테고리명)을 알아야만 계산을 수행할 수 있을 때, 바로 **상관 서브쿼리(Correlated Subquery)**를 사용해야 한다.

여기서 상관(Correlated)의 의미는 메인쿼리와 서브쿼리가 서로 영향을 준다는 뜻이다.

왜 상관 서브쿼리가 필요한가?

이 문제를 해결하려면, `products` 테이블의 각 행을 하나씩 확인하면서 비교 대상을 동적으로 바꿔야 한다.

product_id	name	category	price	비교해야 할 대상
1	프리미엄 게이밍 마우스	전자기기	75000	`SELECT AVG(price) FROM products WHERE category = '전자기기'`
2	기계식 키보드	전자기기	120000	`SELECT AVG(price) FROM products WHERE category = '전자기기'`
3	4K UHD 모니터	전자기기	350000	`SELECT AVG(price) FROM products WHERE category = '전자기기'`
4	관계형 데이터베이스 입문	도서	28000	`SELECT AVG(price) FROM products WHERE category = '도서'`
5	고급 가죽 지갑	패션	150000	`SELECT AVG(price) FROM products WHERE category = '패션'`
6	스마트 워치	전자기기	280000	`SELECT AVG(price) FROM products WHERE category = '전자기기'`

위 표에서 볼 수 있듯이, 우리는 `products` 테이블의 모든 상품을 하나씩 훑어봐야 한다. 그리고 각 상품에 대해, 그 상품이 속한 카테고리의 평균 가격을 **그때그때 계산해서** 현재 상품의 가격과 비교해야 한다.

따라서 다음과 같이 각각의 상품마다 다른 서브쿼리를 실행해야 한다.

```
SELECT AVG(price) FROM products WHERE category = '전자기기' -- product_id:1
SELECT AVG(price) FROM products WHERE category = '전자기기' -- product_id:2
SELECT AVG(price) FROM products WHERE category = '전자기기' -- product_id:3
SELECT AVG(price) FROM products WHERE category = '도서'     -- product_id:4
SELECT AVG(price) FROM products WHERE category = '패션'     -- product_id:5
SELECT AVG(price) FROM products WHERE category = '전자기기' -- product_id:6
```

정리하면 메인쿼리의 각 행마다 다음 서브쿼리를 추가로 실행해야 한다.
그리고 `{category}` 값은 메인쿼리의 각 행마다 다른 `category` 값을 사용해야 한다.

```
SELECT AVG(price) FROM products WHERE category = {category}
```

기존의 비상관 서브쿼리처럼 서브쿼리가 한 번만 실행되어서는 이 문제를 해결할 수 없다. 메인쿼리가 어떤 상품을 보고 있는지에 따라 서브쿼리의 계산 결과가 계속 달라져야 하기 때문이다.

상관 서브쿼리의 개념

상관 서브쿼리는 이름 그대로, **메인쿼리와 서브쿼리가 서로 '연관' 관계를 맺고 동작**하는 서브쿼리다. 서브쿼리가 독립적으로 실행될 수 없고, 메인쿼리의 값을 참조하여 실행되는 것이 특징이다.

상관 서브쿼리의 동작 방식은 기존 서브쿼리와 완전히 다르다.

- **비상관 서브쿼리 (Non-correlated)**: 서브쿼리가 **단 한 번** 실행된 후, 그 결과를 메인쿼리가 사용한다.
- **상관 서브쿼리 (Correlated)**:
  1. 메인쿼리가 먼저 한 행을 읽는다.
  2. 읽혀진 행의 값을 서브쿼리에 전달하여, 서브쿼리가 **실행**된다.
  3. 서브쿼리 결과를 이용해 메인쿼리의 `WHERE` 조건을 판단한다.
  4. 메인쿼리의 다음 행을 읽고, 2-3번 과정을 **반복**한다.

즉, 서브쿼리가 메인쿼리의 행 수만큼 반복 실행될 수 있다. 이제 실제 예제를 통해 이 개념을 확실히 이해해 보자.

상관 서브쿼리로 문제 해결하기

"각 상품별로, 자신이 속한 카테고리의 평균 가격 이상의 상품들을 찾아라."

먼저, 각 카테고리별 평균 가격을 직접 계산해보자.

- **전자기기**: (`75000` + `120000` + `350000` + `280000`) / 4 = 206,250원
- **도서**: `28000`원 (상품이 1개)
- **패션**: `150000`원 (상품이 1개)

우리는 이 값들을 기준으로 각 상품의 가격을 비교해야 한다.

```
SELECT
    product_id,
    name,
    category,
    price
FROM
    products p1
WHERE
    price >= (
        SELECT
            AVG(price)
        FROM
            products p2
        WHERE
            p2.category = p1.category
    );
```

이 쿼리에서 가장 중요한 부분은 서브쿼리 안의 `WHERE p2.category = p1.category` 이다.

- `p1`은 메인쿼리의 `products` 테이블을 가리키는 별칭이다.
- `p2`는 서브쿼리의 `products` 테이블을 가리키는 별칭이다.
- 이 조건문은 "서브쿼리에서 평균을 계산할 때, 메인쿼리가 현재 보고 있는 상품(`p1`)과 동일한 카테고리를 가진 상품들(`p2`)만을 대상으로 하라"는 의미다. 이것이 바로 '연관'의 핵심이다.

쿼리 실행 흐름

데이터베이스가 이 쿼리를 어떻게 처리하는지 단계별로 따라가 보자.

product_id	name	category	price	비교해야 할 대상
1	프리미엄 게이밍 마우스	전자기기	75000	`SELECT AVG(price) FROM products WHERE category = '전자기기'`
2	기계식 키보드	전자기기	120000	`SELECT AVG(price) FROM products WHERE category = '전자기기'`
3	4K UHD 모니터	전자기기	350000	`SELECT AVG(price) FROM products WHERE category = '전자기기'`
4	관계형 데이터베이스 입문	도서	28000	`SELECT AVG(price) FROM products WHERE category = '도서'`
5	고급 가죽 지갑	패션	150000	`SELECT AVG(price) FROM products WHERE category = '패션'`
6	스마트 워치	전자기기	280000	`SELECT AVG(price) FROM products WHERE category = '전자기기'`

- 메인쿼리는 우선 `products` 행들을 읽는다. 그리고 각 행에 대해서 순서대로 다음 로직을 수행한다.

1. 메인쿼리가 `products` (p1)의 첫 행 '프리미엄 게이밍 마우스'(가격 75000, 카테고리 '전자기기')를 읽는다. 이때 `p1.category` 는 '전자기기'다.
2. 이 값을 받아 서브쿼리가 실행된다: `SELECT AVG(price) FROM products p2 WHERE p2.category = '전자기기'` . 결과는 `206250` 이다.
3. 메인쿼리의 `WHERE` 절은 `WHERE 75000 >= 206250` 으로 바뀐다. 이 조건은 거짓(false)이므로 이 행은 결과에 포함되지 않는다.
4. 메인쿼리가 다음 행 '기계식 키보드'(가격 120000, 카테고리 '전자기기')를 읽는다.
5. 서브쿼리가 다시 실행된다. `p1.category` 는 여전히 '전자기기'이므로, `SELECT AVG(price) FROM products p2 WHERE p2.category = '전자기기'` 가 실행되고 결과는 `206250` 이다.
6. `WHERE` 절은 `WHERE 120000 >= 206250` 으로 바뀐다. 거짓이므로 이 행도 포함되지 않는다.
7. 메인쿼리가 다음 행 '4K UHD 모니터'(가격 350000, 카테고리 '전자기기')를 읽는다.

8. 서브쿼리는 또 '전자기기'의 평균 가격인 `206250`을 반환한다.
9. `WHERE` 절은 `WHERE 350000 >= 206250` 으로 바뀐다. 참(true)이므로 이 행은 최종 결과에 포함된다.
10. 메인쿼리가 다음 행 '관계형 데이터베이스 입문'(가격 28000, 카테고리 '도서')을 읽는다. `p1.category`는 이제 '도서'다.
11. 서브쿼리가 실행된다: `SELECT AVG(price) FROM products p2 WHERE p2.category = '도서'`. 결과는 `28000`이다.
12. `WHERE` 절은 `WHERE 28000 >= 28000` 으로 바뀐다. 참이므로 이 행도 결과에 포함된다.
13. 이 과정이 `products` 테이블의 모든 행에 대해 반복된다.

[실행 결과]

product_id	name	category	price
3	4K UHD 모니터	전자기기	350000
4	관계형 데이터베이스 입문	도서	28000
5	고급 가죽 지갑	패션	150000
6	스마트 워치	전자기기	280000

결과를 보면, 각자 자신이 속한 카테고리의 평균 가격(`전자기기`: 206,250원, `도서`: 28,000원, `패션`: 150,000원)과 이상인 상품들만 정확하게 조회된 것을 확인할 수 있다.

상관 서브쿼리2

한 번이라도 주문된 상품 조회하기

우리가 운영하는 쇼핑몰의 수많은 상품 중에 고객들이 어떤 상품을 주로 구매하는지 파악하는 것은 매우 중요하다. 가장 기본적인 분석은 "지금까지 한 번이라도 주문된 적이 있는 상품"이 무엇인지 알아보는 것이다.

`products` 테이블에는 모든 상품 정보가 있고, `orders` 테이블에는 주문 기록이 있다. 이 두 테이블을 함께 사용해서 이 문제를 해결해야 한다.

문제 상황

`products` 테이블에는 있지만, `orders` 테이블에는 한 번도 등장하지 않은 상품, 즉 '재고'로만 남아있는 상품을 제외하고, 실제 주문이 발생한 상품의 이름과 가격을 조회하고 싶다.

직접 확인하는 해결 방법

먼저 각 테이블을 직접 하나하나 눈으로 확인하는 단순한 방법을 사용해보자.

[products 테이블]

```
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

[주문 내역이 있는 상품]

```
SELECT DISTINCT product_id FROM orders;
```

product_id
1
2
3
4

이제 `product_id` 기준으로 1,2,3,4 제품만 찾으면 된다.

[products 테이블]

```
SELECT product_id, name, price
FROM products
WHERE product_id IN (1,2,3,4)
```

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

물론 이렇게 직접 하나하나 나누어 실행하는 것은 좋은 방법이 아니다. 서브쿼리(Subquery)를 사용해서 문제를 해결해 보자. 먼저 `IN` 연산자를 사용해서 해결해보자.

`IN` 을 사용한 해결 방법

`IN` 연산자는 특정 컬럼의 값이 괄호 안에 있는 목록에 포함되는지를 확인하는 역할을 한다. 우리는 이 원리를 이용해 "주문된 상품 ID 목록"을 먼저 구하고, `products` 테이블에서 해당 ID를 가진 상품들을 찾아낼 수 있다.

단계적 접근

1. **주문된 상품 ID 목록 조회**: 먼저 `orders` 테이블에서 어떤 상품들이 주문되었는지 `product_id`를 모두 가져온다. 이때 중복된 ID가 여러 개 있을 수 있으니 `DISTINCT`를 사용해 고유한 `product_id` 목록을 만든다.
2. **상품 정보 조회**: `products` 테이블에서 `product_id`가 1단계에서 만든 목록 안에 `IN`하는 상품들만 조회한다.

쿼리 작성

이 두 단계를 하나의 쿼리로 합치면 다음과 같다.

```
SELECT
    product_id,
    name,
    price
```

```
FROM
    products
WHERE
    product_id IN (SELECT DISTINCT product_id FROM orders);
```

쿼리 실행 흐름

1. 데이터베이스는 먼저 서브쿼리인 `SELECT DISTINCT product_id FROM orders`를 실행한다.
2. `orders` 테이블에 기록된 모든 주문을 확인하여, 주문된 상품들의 고유한 ID 목록을 만든다.
   - `orders` 테이블에는 `product_id`가 `1, 4, 2, 4, 3, 1, 1`인 주문들이 있다.
   - `DISTINCT`를 통해 중복이 제거된 최종 목록은 `(1, 2, 3, 4)`가 된다.
3. 이제 메인쿼리가 실행된다.
   `SELECT product_id, name, price FROM products WHERE product_id IN (1, 2, 3, 4);`
4. `products` 테이블의 모든 상품을 하나씩 확인하며 `product_id`가 `(1, 2, 3, 4)` 목록에 포함되는지 검사한다.
   - `product_id`가 1인 '프리미엄 게이밍 마우스'는 목록에 있으므로 결과에 포함된다.
   - ...
   - `product_id`가 5인 '고급 가죽 지갑'은 목록에 없으므로 결과에서 제외된다.
   - `product_id`가 6인 '스마트 워치'도 목록에 없으므로 결과에서 제외된다.

[실행 결과]

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

`IN`을 사용하면 이처럼 원하는 결과를 직관적으로 얻을 수 있다.
이 방법은 대부분의 경우에 아주 효율적인 방법이다. 하지만 실무에서는 이 방식이 항상 최선은 아니다. 왜 그럴까?

`EXISTS`를 사용한 더 효율적인 방법

`IN` 방식은 `orders` 테이블이 매우 클 경우, 즉 주문량이 수백만, 수천만 건에 달하면 성능 문제를 일으킬 수 있다.

`orders`를 조회하는 서브쿼리가 반환하는 지금까지 주문한 `product_id` 목록 전체를 메모리에 저장한 뒤, 메인쿼리의 각 행과 비교해야 하기 때문이다.

이럴 때 `EXISTS`를 사용하면 효율적으로 쿼리를 실행할 수 있다.
`EXISTS`는 서브쿼리가 반환하는 **결과값 자체에는 관심이 없고**, 오직 서브쿼리의 결과로 **행이 하나라도 존재하는지 여부**만 체크한다.

EXISTS 연산

- 서브쿼리 결과 행이 1개 이상이면 `TRUE`
- 서브쿼리 결과 행이 0개이면 `FALSE`

`EXISTS`는 서브쿼리가 **결과를 반환하는지 여부(Existence)**만 확인한다. 서브쿼리가 단 하나의 행이라도 반환하면 `TRUE`, 아무 행도 반환하지 않으면 `FALSE`가 된다.

[EXISTS SQL]

```
SELECT
    product_id,
    name,
    price
FROM
    products p
WHERE EXISTS (
    SELECT 1
    FROM orders o
    WHERE o.product_id = p.product_id
);
```

여기서 `p`와 `o`는 각각 `products`와 `orders` 테이블의 별칭이다. `EXISTS` 안의 서브쿼리를 보면 `WHERE o.product_id = p.product_id` 라는 조건이 있다. 여기서는 상관 서브쿼리를 사용했다. 서브쿼리가 독립적으로 실행되지 않고 메인쿼리의 `p` 테이블 값에 의존하여 실행된다는 의미이다.

또한, 서브쿼리가 `SELECT 1`을 사용하는 것을 볼 수 있다. `EXISTS`는 결과 데이터가 무엇인지는 전혀 신경 쓰지 않고, 행이 존재하는지 여부만 보기 때문에 관례적으로 `SELECT 1`과 같이 상수를 사용해 불필요한 데이터 조회를 피한다.

[실행 결과]

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

쿼리 실행 흐름

`EXISTS`의 실행 흐름은 `IN`과 완전히 다르다.

1. 메인쿼리가 `products`(p) 테이블의 첫 번째 행인 '프리미엄 게이밍 마우스'(`p.product_id = 1`)를 읽는다.
2. 이 `p.product_id` 값을 가지고 서브쿼리가 실행된다:
   `SELECT 1 FROM orders o WHERE o.product_id = 1`
3. `orders` 테이블에는 `product_id`가 1인 주문이 3개 존재한다. 데이터베이스는 조건을 만족하는 **첫 번째 행을 찾자마자** 더 이상 테이블을 탐색하지 않고, 서브쿼리가 결과를 반환할 수 있다고 판단한다. (예를 들어서 `product_id`가 1인 주문이 10000개 존재하더라도 `EXISTS`는 첫 번째 행만 찾으면 바로 `TRUE`를 반환한다. 따라서 나머지 9999개를 찾지 않아도 된다.)
4. `EXISTS`는 `TRUE`가 된다.
5. `WHERE TRUE` 조건이 충족되었으므로, '프리미엄 게이밍 마우스'는 최종 결과에 포함된다.
6. ... `p.product_id = 2,3,4`의 같은 내용이 반복된다. 이 내용은 최종 결과에 포함된다.
7. 메인쿼리가 `products` 테이블의 다음 행인 '고급 가죽 지갑'(`p.product_id = 5`)을 읽는다.
8. 다시 서브쿼리가 실행된다:
   `SELECT 1 FROM orders o WHERE o.product_id = 5`
9. `orders` 테이블을 탐색했지만 `product_id`가 5인 주문은 존재하지 않는다. 서브쿼리는 아무런 행도 반환하지 못한다.
10. `EXISTS`는 `FALSE`가 된다.
11. `WHERE FALSE` 조건이므로, '고급 가죽 지갑'은 최종 결과에서 제외된다.
12. 이후에 `p.product_id = 6`도 `WHERE FALSE` 조건이 되면서 최종 결과에서 제외된다.

결과적으로 이 과정을 `products` 테이블의 모든 행에 대해 반복한다.

`IN` vs. `EXISTS`: 실무에서는?

구분	IN	EXISTS

실행 방식	서브쿼리를 먼저 실행해 결과 목록을 만든 후, 메인쿼리에서 사용	메인쿼리의 각 행에 대해 서브쿼리를 실행하여 조건 확인
특징	서브쿼리 결과가 작을 때 직관적이고 빠를 수 있음	상관 서브쿼리. 서브쿼리 테이블이 클 때 효율적
최적화	`orders` 테이블 전체를 스캔해야 할 수 있음	조건을 만족하는 첫 행만 찾으면 스캔을 멈춤

결론적으로, 서브쿼리의 대상이 되는 테이블(`orders`)이 크다면 `EXISTS`가 유리하다. `IN`은 목록 전체를 비교해야 하지만, `EXISTS`는 조건을 만족하는 데이터를 '발견'하는 즉시 다음으로 넘어가기 때문이다.

`NOT EXISTS`: 존재하지 않음을 확인하기

반대로, 특정 조건의 데이터가 존재하지 않는 것을 확인하고 싶을 때는 `NOT EXISTS`를 사용한다.

문제 상황: "한 번도 주문된 적이 없는, 이른바 '악성 재고' 상품을 찾아라."

```
SELECT
    product_id,
    name,
    price,
    stock_quantity
FROM
    products p
WHERE NOT EXISTS (
    SELECT 1
    FROM orders o
    WHERE o.product_id = p.product_id
);
```

이 쿼리는 위 `EXISTS` 예제와 정반대로 동작하여, 서브쿼리의 결과가 0건일 때 `TRUE`를 반환한다.

[실행 결과]

product_id	name	price	stock_quantity
5	고급 가죽 지갑	150000	15
6	스마트 워치	280000	40

상관 서브쿼리와 성능

상관 서브쿼리는 복잡한 로직을 매우 직관적으로 표현할 수 있게 해주지만, **성능에 주의**해야 한다. 메인쿼리의 행 수만큼 서브쿼리가 반복 실행될 수 있기 때문에, 메인쿼리가 다루는 데이터의 양이 많아지면 쿼리 전체의 성능이 급격히 저하될 수 있다.

많은 경우, 상관 서브쿼리는 `JOIN` (특히 `LEFT JOIN`과 `GROUP BY`)으로 동일한 결과를 얻도록 재작성할 수 있으며, 데이터베이스 옵티마이저가 `JOIN`을 더 효율적으로 처리하는 경우가 많다.

하지만 `EXISTS`는 특정 조건에 맞는 데이터가 있는지 '확인만 하고 넘어가는' 특성 덕분에, `IN` 이나 `JOIN` 보다 훨씬 효율적으로 동작하는 상황도 많다.

결론적으로 상관 서브쿼리는 성능 이슈를 인지하고, `JOIN`으로 표현하기 너무 복잡하거나 `EXISTS`를 통해 더 효율적인 실행이 가능할 때 적절히 사용하는 것이 중요하다.

지금까지 우리는 서브쿼리를 `WHERE` 절에서 사용하여, 필터링할 조건을 동적으로 만드는 방법을 배웠다. 하지만 서브쿼리의 활약은 여기서 그치지 않는다. 만약 서브쿼리를 `SELECT` 절 안으로 가져온다면 어떻게 될까? 각 행마다 새로운 정보를 계산해서 보여주는, 전혀 다른 방식의 활용이 가능해진다.

다음 시간에는 `SELECT` 절에서 활약하는 **스칼라 서브쿼리**에 대해 알아보겠다.

SELECT 서브쿼리

우리는 지금까지 `WHERE` 절에서 서브쿼리를 사용하여 필터링할 조건을 동적으로 만들어내는 방법을 배웠다. 서브쿼리가 반환하는 값(들)을 기준으로 메인쿼리가 보여줄 행을 걸러내는 방식이었다.

이제 시선을 바꿔보자. 만약 서브쿼리를 `SELECT` 절 안으로 가져온다면 어떻게 될까? 서브쿼리는 더 이상 필터가 아닌, 그 자체가 하나의 **'컬럼'**처럼 동작하게 된다.

참고로 SELECT 절에서는 단일 값(하나의 행, 하나의 컬럼)을 반환하는 **스칼라 서브쿼리**를 사용해야 한다.

비상관 서브쿼리

먼저, 바깥쪽 쿼리에 전혀 영향을 받지 않는 독립적인 서브쿼리부터 살펴보자.

문제 상황: "모든 상품 목록을 조회하는데, 각 상품의 가격과 함께 **전체 상품의 평균 가격**을 모든 행에 함께 표시해서 개별 상품 가격이 평균과 얼마나 차이 나는지 비교해보고 싶다."

이 경우, '전체 상품의 평균 가격'은 어떤 특정 상품 행에 종속되는 값이 아니라, 모든 상품에 대해 동일하게 적용되는 고정된 값이다. 이런 경우에 간단한 스칼라 서브쿼리를 사용할 수 있다.

먼저, 전체 상품의 평균 가격을 구하는 쿼리는 다음과 같다.

```
SELECT AVG(price) FROM products;
```

[실행 결과]

AVG(price)
167166.6667

이제 이 쿼리를 `SELECT` 절 안에 그대로 넣어보자.

```
SELECT
    name,
    price,
    (SELECT AVG(price) FROM products) AS avg_price
FROM
    products;
```

- `avg_price`는 스칼라 서브쿼리이다. `SELECT`에 사용하는 서브쿼리는 **스칼라 서브쿼리**만 가능하다.

[실행 결과]

name	price	avg_price

프리미엄 게이밍 마우스	75000	167166.6667
기계식 키보드	120000	167166.6667
4K UHD 모니터	350000	167166.6667
관계형 데이터베이스 입문	28000	167166.6667
고급 가죽 지갑	150000	167166.6667
스마트 워치	280000	167166.6667

쿼리 실행 흐름

1. 데이터베이스는 메인쿼리를 실행하기 전에, `SELECT` 절의 스칼라 서브쿼리를 **단 한 번** 먼저 실행한다.
   `(SELECT AVG(price) FROM products)` ➜ `167166.6667`
2. 데이터베이스는 이 계산된 값을 기억해 둔다.
3. 메인쿼리(`SELECT name, price, ... FROM products`)가 실행된다.
4. `products` 테이블의 각 행을 가져올 때마다, `avg_price` 컬럼에 미리 계산해 둔 값(`167166.6667`)을 그대로 추가한다.

이처럼 서브쿼리가 외부 쿼리의 컬럼을 참조하지 않아 독립적으로 실행될 수 있는 경우를 앞서 배웠듯이 **비상관 서브쿼리(Non-correlated Subquery)**라고 한다. 서브쿼리가 먼저 한 번 실행되고, 그 결과가 메인쿼리의 모든 행에 재사용되는 방식이다.

결과를 보면 모든 상품 행에 동일한 전체 평균 가격이 `avg_price` 컬럼으로 잘 추가된 것을 확인할 수 있다.

상관 서브쿼리

비상관 서브쿼리는 유용하지만, `SELECT` 절의 스칼라 서브쿼리의 진정한 강력함은 **메인쿼리의 각 행과 상호작용**할 때 드러난다.

새로운 문제 상황을 보자.
"전체 상품 목록을 조회하면서, 각 상품별로 총 몇 번의 주문이 있었는지 '총 주문 횟수'를 함께 보여주고 싶다."

이제는 '전체 평균'처럼 고정된 값이 아니라, '프리미엄 게이밍 마우스'의 주문 횟수, '기계식 키보드'의 주문 횟수 등 각 상품 행에 따라 **계산 결과가 달라져야 하는 값**이 필요하다.

product_id	name	price	order_count

1	프리미엄 게이밍 마우스	75000	`SELECT COUNT(*) FROM orders o WHERE o.product_id = 1`
2	기계식 키보드	120000	`SELECT COUNT(*) FROM orders o WHERE o.product_id = 2`
3	4K UHD 모니터	350000	`SELECT COUNT(*) FROM orders o WHERE o.product_id = 3`
4	관계형 데이터베이스 입문	28000	`SELECT COUNT(*) FROM orders o WHERE o.product_id = 4`
5	고급 가죽 지갑	150000	`SELECT COUNT(*) FROM orders o WHERE o.product_id = 5`
6	스마트 워치	280000	`SELECT COUNT(*) FROM orders o WHERE o.product_id = 6`

이 문제를 해결하기 위해, 우리는 `products` 테이블의 각 상품 행을 하나씩 보면서, 그 상품이 `orders` 테이블에서 몇 번이나 주문되었는지를 개별적으로 세어야 한다.

여기서 각 상품별 총 주문 횟수를 구하려면 다음과 같이 주문 번호가 서로 다른 서브쿼리를 실행해야 한다.

```
SELECT COUNT(*) FROM orders o WHERE o.product_id = 1;
SELECT COUNT(*) FROM orders o WHERE o.product_id = 2;
SELECT COUNT(*) FROM orders o WHERE o.product_id = 3;
SELECT COUNT(*) FROM orders o WHERE o.product_id = 4;
SELECT COUNT(*) FROM orders o WHERE o.product_id = 5;
SELECT COUNT(*) FROM orders o WHERE o.product_id = 6;
```

정리하면 메인쿼리의 각 행마다 다음 서브쿼리를 추가로 실행해야 한다.
그리고 `{product_id}` 값은 메인쿼리의 각 행마다 다른 `product_id` 값을 사용하면 된다.

```
SELECT COUNT(*) FROM orders o WHERE o.product_id = {product_id}
```

서브쿼리의 중요한 특징은, 바깥쪽 메인쿼리의 각 행마다 **개별적으로, 반복적으로 실행될 수 있다**는 점이다. 서브쿼리가 메인쿼리의 컬럼 값을 참조하는 관계를 가질 때, 이를 **상관 서브쿼리(Correlated Subquery)**라고 부른다.

이제 실제 쿼리를 작성해 보자.
메인쿼리는 `products` 테이블에서 상품 정보를 가져오는 것이다. 그리고 `SELECT` 절에 '총 주문 횟수'를 계산하는 서브쿼리를 하나의 컬럼처럼 추가한다.

```
SELECT
    p.product_id,
    p.name,
    p.price,
    (SELECT COUNT(*) FROM orders o WHERE o.product_id = p.product_id) AS
order_count
FROM
    products p;
```

여기서 가장 중요한 부분은 서브쿼리 안의 `WHERE o.product_id = p.product_id` 조건이다.
`p.product_id`는 메인쿼리(`FROM products p`)가 현재 처리하고 있는 행의 `product_id` 값을 의미한다.

[실행 결과]

product_id	name	price	order_count
1	프리미엄 게이밍 마우스	75000	3
2	기계식 키보드	120000	1
3	4K UHD 모니터	350000	1
4	관계형 데이터베이스 입문	28000	2
5	고급 가죽 지갑	150000	0
6	스마트 워치	280000	0

[쿼리 실행 흐름]

1. 메인쿼리가 `products` 테이블의 첫 번째 행, '프리미엄 게이밍 마우스'(`p.product_id`=1)를 읽는다.
2. 이 행의 `order_count` 값을 계산하기 위해 스칼라 서브쿼리가 실행된다. 이때 `p.product_id`에는 `1`이 전달된다.
   `(SELECT COUNT(*) FROM orders o WHERE o.product_id = 1)`
3. 이 서브쿼리는 `orders` 테이블에서 `product_id`가 1인 주문을 세어 `3`이라는 단일 값을 반환한다.
4. 첫 번째 행의 `order_count` 컬럼에는 `3`이 기록된다.

5. 메인쿼리가 두 번째 행, '기계식 키보드'(`p.product_id`=2)를 읽는다.
6. 다시 스칼라 서브쿼리가 실행된다. 이번에는 `p.product_id`에 `2`가 전달된다.
   `(SELECT COUNT(*) FROM orders o WHERE o.product_id = 2)`
7. 서브쿼리는 `1`을 반환하고, 두 번째 행의 `order_count`에는 `1`이 기록된다.
8. 이 과정이 `products` 테이블의 모든 행에 대해 반복된다.

결과를 보면 각 상품별로 정확한 주문 횟수가 새로운 컬럼처럼 조회된 것을 볼 수 있다. 한 번도 팔리지 않은 상품의 주문 횟수는 `0`으로 표시된다.

실무 팁: 성능에 주의하라

스칼라 서브쿼리는 이처럼 `JOIN`으로는 표현하기 복잡한 로직을 직관적으로 표현할 수 있게 해주지만, 강력한 만큼 주의해서 사용해야 한다.

가장 큰 단점은 **성능 저하**의 가능성이다. 특히 상관 서브쿼리는 메인쿼리가 반환하는 행의 수만큼 서브쿼리가 반복 실행되기 때문이다. 만약 `products` 테이블에 100만 개의 상품이 있다면, 주문 횟수를 알기 위해 `COUNT(*)` 쿼리가 100만 번이나 실행되는 셈이다. 이는 데이터베이스에 엄청난 부하를 줄 수 있다.

사실 오늘의 문제 상황은 `LEFT JOIN`과 `GROUP BY`를 사용해서도 해결할 수 있으며, 대부분의 경우 데이터베이스 옵티마이저가 `JOIN`을 더 효율적으로 처리하여 성능이 더 좋다.

```sql
-- JOIN으로 해결하는 방법
SELECT p.product_id, p.name, p.price, COUNT(o.order_id) AS order_count
FROM products p
LEFT JOIN orders o ON p.product_id = o.product_id
GROUP BY p.product_id, p.name, p.price;
```

그럼에도 불구하고 스칼라 서브쿼리는 `JOIN`이 너무 복잡해지거나, 완전히 다른 테이블에서 간단한 정보 하나만 조회해 올 때 코드를 훨씬 명료하게 만들어주는 장점이 있어 적재적소에 사용하면 매우 유용하다.

우리는 지금까지 서브쿼리를 `WHERE` 절과 `SELECT` 절에서 사용하는 법을 배웠다. 이제 마지막 남은 장소는 `FROM` 절이다. 쿼리의 결과를 하나의 가상 테이블로 취급하고, 그 가상 테이블로부터 다시 데이터를 조회하는 방법은 없을까?

다음 시간에는 `FROM` 절에서 활약하는 서브쿼리, **인라인 뷰(Inline View)**에 대해 알아보겠다.

테이블 서브쿼리

지금까지 서브쿼리가 `WHERE` 절에서는 '동적 필터'로, `SELECT` 절에서는 '새로운 컬럼'으로 활약하는 모습을 보았다.
이제 서브쿼리가 자리할 수 있는 마지막 주요 위치, 바로 `FROM` 절에 대해 알아볼 시간이다.

`FROM` 절에 위치하는 서브쿼리는, 그 실행 결과가 마치 **하나의 독립된 가상 테이블**처럼 사용되기 때문에 테이블 서브쿼리라 한다.
쿼리 내에서 '인라인(inline)'으로 즉석에서 정의되는 '뷰(View, 가상 테이블)'와 같다고 해서 **인라인 뷰(Inline View)**라고도 부른다.

테이블 서브쿼리의 가장 큰 특징은, 우리가 지금까지 배운 복잡한 `SELECT` 문(집계, 그룹핑, 조인 등)의 결과를 하나의 명확한 데이터 집합으로 먼저 만들어 놓고, 그 집합을 대상으로 다시 한번 `SELECT`를 할 수 있게 해준다는 점이다.

이 개념을 제대로 이해하기 위해, `GROUP BY`만으로는 한 번에 풀기 어려운 까다로운 문제 상황을 살펴보자.
"각 상품 카테고리별로, 가장 비싼 상품의 이름과 가격을 조회하고 싶다."

이 문제를 보면 카테고리별로 묶어야 하니까 `GROUP BY`를 사용하고, 가장 비싼 상품의 가격은 `MAX()`를 사용하면 될 것 같다는 생각이 들 것이다. 하지만 문제는 이름이다.

```
-- name이 빠짐
SELECT category, MAX(price)
FROM products
GROUP BY category;
```

[실행 결과]

category	MAX(price)
전자기기	350000
도서	28000
패션	150000

여기에 이름(`name`)을 추가해보자.

```
-- 잘못된 쿼리의 예
SELECT category, name, MAX(price)
FROM products
GROUP BY category;
```

[실행 결과]

```
Error Code: 1055. Expression #2 of SELECT list is not in GROUP BY clause and
contains nonaggregated column 'my_shop2.products.name' which is not
functionally dependent on columns in GROUP BY clause; this is incompatible
with sql_mode=only_full_group_by
```

`products.name` 컬럼은 `GROUP BY` 절에 없기 때문에 사용할 수 없다는 오류이다.

위 쿼리를 실행하면, 데이터베이스는 `GROUP BY`로 묶인 `category`별로 `MAX(price)`는 정확하게 계산할 수 있지만, `name`은 해당 그룹의 여러 상품명 중에 어떤 것을 보여줄지 선택할 수 없다. 따라서 오류가 발생한다.

예를 들어서 `전자기기` 카테고리만 해도 각각 다른 4개의 상품명이 존재한다. 이 중에 어떤 상품명 하나를 선택해야 할지 적절한 기준이 없으므로 오류가 발생한다.

```
SELECT name, category, price FROM products
WHERE category = '전자기기';
```

name	category	price
프리미엄 게이밍 마우스	전자기기	75000
기계식 키보드	전자기기	120000
4K UHD 모니터	전자기기	350000
스마트 워치	전자기기	280000

우리가 원하는 것은 '각 카테고리별 최고가'를 먼저 알아낸 뒤, 그 가격과 일치하는 '바로 그 상품'을 찾는 것이다. 이 2단계 로직을 구현하는 데 인라인 뷰가 적절한 해법을 제시한다.

인라인 뷰를 이용한 2단계 접근법

1단계 (인라인 뷰): 카테고리별 최고가격을 미리 구한다.
먼저, 카테고리별로 가장 높은 가격이 얼마인지 알아내는 쿼리를 작성한다. 이 쿼리가 바로 인라인 뷰, 즉 우리의 가상 테이블이 될 것이다.

```
SELECT category, MAX(price) AS max_price
 FROM products
 GROUP BY category
```

category	max_price
도서	28000
전자기기	350000
패션	150000

이 쿼리를 실행하면 '전자기기', '도서', '패션' 각 카테고리의 최고가격을 담은 3줄짜리 결과가 나온다. 이제 이 결과를 `category_max_price`라는 이름의 가상 테이블이라고 생각하자.

2단계 (메인쿼리): 원본 테이블과 가상 테이블을 조인한다.
이제 원본 `products` 테이블과, 우리가 방금 만든 가상 테이블(`category_max_price`)을 조인한다.
연결 조건은 두 가지다.

1. 카테고리 이름이 같아야 한다 (`p.category = cmp.category`).
2. 상품 가격이 그 카테고리의 최고가와 같아야 한다 (`p.price = cmp.max_price`).

이 두 조건을 모두 만족하는 상품만이 '카테고리별 최고가 상품'이라는 것을 보장할 수 있다.

최종 쿼리 작성

이제 두 단계를 합쳐 최종 쿼리를 완성해 보자. `FROM` 절에 들어가는 서브쿼리에는 **반드시 별칭(Alias)을 붙여줘야 한다**는 점을 잊지 말자. 여기서는 `cmp`(category max price)라는 별칭을 사용했다.

```
SELECT
```

```
    p.product_id,
    p.name,
    p.price
FROM
    products p
JOIN
    (SELECT
         category,
         MAX(price) AS max_price
     FROM
         products
     GROUP BY
         category) AS cmp
ON
    p.category = cmp.category AND p.price = cmp.max_price;
```

[실행 결과]

product_id	name	price
3	4K UHD 모니터	350000
4	관계형 데이터베이스 입문	28000
5	고급 가죽 지갑	150000

실행 결과 분석

[기준 테이블 - products]

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

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

5	고급 가죽 지갑	패션	150000
6	스마트 워치	전자기기	280000

[조인 대상 테이블 - 인라인 뷰]

```
SELECT category, MAX(price) AS max_price
 FROM products
 GROUP BY category
```

category	max_price
도서	28000
전자기기	350000
패션	150000

쿼리 실행 흐름

1. 데이터베이스는 `FROM` 절의 서브쿼리(인라인 뷰)를 먼저 실행하여, `cmp` 라는 임시 테이블을 메모리에 생성한다. 이 테이블에는 각 카테고리와 그 카테고리의 최고 가격이 들어있다.
2. 그다음, 메인쿼리가 실행된다. `products` 테이블(별칭 `p`)과 방금 생성된 임시 테이블 `cmp` 를 `INNER JOIN` 한다.
3. `ON` 절에 명시된 두 가지 조건(카테고리 일치, 가격 일치)을 모두 만족하는 행만 최종 결과로 선택된다.

결과를 보면 `GROUP BY`의 함정을 피하고, 각 카테고리별로 가장 비싼 상품의 정보를 정확하게 찾아냈다.
이처럼 인라인 뷰는 복잡한 데이터를 단계적으로 가공해야 할 때, 특히 집계된 결과를 가지고 다시 한번 조인이나 필터링을 수행해야 할 때 매우 유용하다.

> **FROM 절의 상관 서브쿼리 - LATERAL**
> `FROM` 절에서 상관 서브쿼리를 사용하려면 `LATERAL`이라는 특별한 키워드를 사용해야 한다. 이 기능은 너무 복잡하고, 성능도 잘 나오지 않는 문제가 있어서 실무에서는 잘 사용하지 않는 편이다. 따라서 여기서는 따로 설명하지 않겠다. `FROM` 절에 상관 서브쿼리를 꼭 사용해야 하는 특별한 일이 있다면 `LATERAL` 키워드를 검색해보자.

지금까지 조인과 서브쿼리 두 가지 핵심 기능을 학습하면서 아마도 어? 이 기능은 두 가지 방법으로 모두 풀 수 있을 것 같은데? 라는 생각이 들었을 것이다. 다음 시간에는 둘의 특징에 대해 자세히 알아보자.

서브쿼리 vs JOIN

지금까지 `JOIN`과 서브쿼리라는 두 가지 강력한 기술을 배웠다. 그러면서 아마도 이런 의문이 들었을 것이다. "어? 어떤 문제는 두 가지 방법으로 모두 풀 수 있네?" 맞다. 실제로 많은 문제는 `JOIN`으로도, 서브쿼리로도 해결할 수 있다. 그렇다면 우리는 무엇을 선택해야 할까? 성능과 가독성 측면에서 둘은 어떤 차이가 있을까?

문제 상황: "서울에 거주하는 모든 고객들의 주문 목록을 조회해라."

해결 방법 1: 서브쿼리 사용

서브쿼리를 이용한 접근법은 우리의 사고 흐름과 매우 유사하다.

1. 먼저, 서울에 사는 고객들의 `user_id` 목록을 찾는다 (`users` 테이블).
2. 그다음, 이 `user_id` 목록에 포함된 `order_id`를 가진 주문들을 찾는다 (`orders` 테이블).

이 논리를 그대로 쿼리로 옮기면 다음과 같다.

```sql
SELECT o.order_id, o.user_id, o.product_id, o.order_date
FROM orders o
WHERE o.user_id IN (SELECT user_id FROM users WHERE address LIKE '서울%');
```

이 쿼리는 읽기가 매우 쉽다. "`users` 테이블에서 `address`가 '서울'인 고객 `user_id` 목록 안에(`IN`), `user_id`가 포함된 `orders`를 찾아줘" 라고 말하는 것과 같다.

[실행 결과]

order_id	user_id	product_id	order_date
1	1	1	2025-06-10 10:00:00
2	1	4	2025-06-10 10:05:00
4	3	4	2025-06-12 09:00:00
6	5	1	2025-06-16 18:00:00

해결 방법 2: JOIN 사용

`JOIN`을 이용한 접근법은 필요한 테이블들을 일단 모두 연결한 뒤, 원하는 조건을 필터링하는 방식이다.

```sql
SELECT o.order_id, o.user_id, o.product_id, o.order_date
FROM orders o
JOIN users u ON o.user_id = u.user_id
WHERE u.address LIKE '서울%';
```

이 쿼리는 "주문(`orders`)과 고객(`users`) 테이블을 `user_id`로 연결한 다음, 그중에서 고객 주소가 '서울'인 데이터만 걸러줘" 라고 말하는 것과 같다. 결과는 당연히 서브쿼리를 사용했을 때와 동일하다.

order_id	user_id	product_id	order_date
1	1	1	2025-06-10 10:00:00
2	1	4	2025-06-10 10:05:00
4	3	4	2025-06-12 09:00:00

6	5	1	2025-06-16 18:00:00

성능 vs 가독성: 실무 가이드

동일한 결과를 내는 두 가지 방법. 그렇다면 우리는 무엇을 선택해야 할까?

성능 (Performance)

일반적으로, 데이터베이스는 `JOIN`이 서브쿼리보다 성능이 더 좋거나 최소한 동일한 경우가 많다.

왜 그럴까? 그 비밀은 데이터베이스의 '두뇌' 역할을 하는 **쿼리 옵티마이저(Query Optimizer)**에 있다.

- `JOIN` 구문은 옵티마이저에게 더 많은 정보를 제공한다. "A와 B 테이블을 특정 조건으로 연결해야 한다"는 전체 그림을 미리 보여주기 때문에, 옵티마이저는 인덱스를 어떻게 활용하고 어떤 테이블을 먼저 읽을지 등 가장 효율적인 실행 계획을 선택할 수 있는 더 넓은 선택지를 갖는다.
- 반면, 서브쿼리는 (특히 과거의 데이터베이스에서는) 단계적으로 실행되는 경우가 많았다. 서브쿼리를 먼저 실행해서 나온 결과를 메모리에 담아두고, 그 다음 메인쿼리가 그 결과를 참조하는 방식으로 동작하여 비효율을 야기할 수 있었다.

하지만! 요즘 데이터베이스의 옵티마이저는 매우 똑똑해져서, 우리가 작성한 예제처럼 간단한 `IN` 서브쿼리는 내부적으로 최적의 `JOIN` 구문으로 자동 변환해서 실행하는 경우가 많다. 따라서 위 예제의 두 쿼리는 사실상 동일한 성능을 낼 확률이 높다.
참고로 이런 최적화는 항상 가능한 것은 아니기 때문에 쿼리 실행 계획 등을 확인하는 것이 좋다.

> **쿼리 옵티마이저와 쿼리 실행 계획에 대한 자세한 내용은 데이터베이스 성능 최적화 강의에서 다룬다.**

가독성 (Readability)

가독성은 주관적인 영역이지만, 쿼리의 유지보수 측면에서 성능만큼이나 중요하다.

- **서브쿼리**는 쿼리의 논리적 단계를 명확하게 구분해 주어, 복잡한 로직을 더 이해하기 쉽게 만들어주는 경우가 많다.
- `JOIN`은 쿼리에 필요한 모든 데이터 소스를 한눈에 보여주고, 여러 테이블의 컬럼을 함께 조회해야 할 때는 구조적으로 더 깔끔하다.

최종 결론: 언제 무엇을 써야 할까?

정답은 없다. 하지만 실무에서 적용할 수 있는 가이드라인은 다음과 같다.

1. **`JOIN`을 우선적으로 고려하라.**
   일반적인 성능 우위와 범용성을 고려할 때, 문제를 해결할 방법을 `JOIN`에서 먼저 찾아보는 것이 좋은 출발점이다.

2. **`JOIN`으로 표현하기 너무 복잡하거나, 서브쿼리의 가독성이 훨씬 좋다면 서브쿼리를 사용하라.**
   성능이 아주 중요한 쿼리가 아니라면, 동료가 이해하기 쉬운 코드를 작성하는 것이 장기적으로 더 가치 있을 수 있다. 특히 인라인 뷰를 사용해야만 깔끔하게 풀리는 문제는 서브쿼리가 정답이다.

3. **`EXISTS`를 활용하라.**
   `IN` 서브쿼리의 대안으로, `EXISTS` 라는 서브쿼리 연산자도 있다. `EXISTS`는 서브쿼리의 결과값이 존재하는지 여부만 체크하기 때문에, 특정 상황에서 더 효율적으로 동작하기도 한다.

4. **성능이 의심될 때는 반드시 측정하라.**
   가장 중요한 원칙이다. 추측하지 말고, `EXPLAIN`과 같은 도구를 사용해 데이터베이스가 어떻게 쿼리를 실행하는지 계획을 분석하고, 실제 실행 시간을 측정하여 더 나은 방법을 선택해야 한다.

`JOIN`과 서브쿼리는 대립하는 기술이 아니라, 데이터라는 재료를 요리하는 두 가지 필수 도구다. 각각의 장단점을 이해하고 상황에 맞게 꺼내 사용할 수 있어야 한다.

문제와 풀이

문제1: 가장 비싼 상품 조회하기

[문제]
`products` 테이블에서 가격이 가장 비싼 상품의 `product_id`, `name`, `price`를 조회해라.
`WHERE` 절에 스칼라 서브쿼리를 사용하여 문제를 해결해야 한다.

[실행 결과]

상품ID	상품명	가격

3	4K UHD 모니터	350000

[정답]

```
SELECT
    product_id AS 상품ID,
    name AS 상품명,
    price AS 가격
FROM
    products
WHERE
    price = (SELECT MAX(price) FROM products);
```

문제2: 동일 상품 주문 정보 조회하기

[문제]

`order_id`가 1인 주문과 동일한 상품을 주문한 다른 모든 주문의 `order_id`, `user_id`, `order_date`를 조회해라.

스칼라 서브쿼리를 활용해야 한다.

[실행 결과]

주문ID	고객ID	주문일시
6	5	2025-06-16 18:00:00
7	2	2025-06-17 12:00:00

[정답]

```sql
SELECT
    order_id AS 주문ID,
    user_id AS 고객ID,
    order_date AS 주문일시
FROM
    orders
WHERE
    product_id = (
        SELECT product_id
        FROM orders
        WHERE order_id = 1
    )
AND order_id != 1;
```

문제3: 고객별 총 주문 횟수 조회하기

[문제]

각 고객(`users`)별로 총 몇 번의 주문을 했는지 '총주문횟수'를 이름과 함께 조회해라. 정렬은 `user_id` 오름차순이다.

한 번도 주문하지 않은 고객도 결과에 포함되어야 한다. `SELECT` 절에 상관 서브쿼리를 사용하여 해결해야 한다.

[실행 결과]

고객명	총주문횟수
션	2
네이트	2
세종대왕	1
이순신	1
마리 퀴리	1
레오나르도 다빈치	0

[정답]

```
SELECT
    name AS 고객명,
    (
        SELECT COUNT(*)
        FROM orders
        WHERE o.user_id = u.user_id
    ) AS 총주문횟수
FROM
    users u
ORDER BY
    u.user_id;
```

정리

서브쿼리 종류와 특징

서브쿼리는 반환하는 **행과 컬럼의 수**에 따라 종류가 나뉘며, 사용되는 **위치와 연산자**에 따라 그 역할이 결정된다.

구분	반환 형태	주요 사용 위치	연산자 / 구문	명칭	핵심 용도
단일 컬럼	단일 행	`SELECT`, `WHERE`, `HAVING`	`=`, `>`, `<` 등	- **스칼라 서브쿼리** (Scalar Subquery) - 단일 값 서브쿼리	**단일 값**이 필요한 모든 곳 (값 비교, 특정 값 표시)
	다중 행	`WHERE`, `HAVING`	`IN`, `ANY`, `ALL`	- **다중 행 서브쿼리** - 값 목록 서브쿼리 - 리스트 서브쿼리 - 컬럼 서브쿼리	**값 목록(List)**과 비교

다중 컬럼	단일 행	`WHERE`, `HAVING`	`(c1, c2) = ...`	- **다중 컬럼 서브쿼리** - 행 서브쿼리	여러 컬럼 값을 **정확히 1:1로 비교** (쌍 비교)
	다중 행	`WHERE`, `HAVING`	`(c1, c2) IN ...`	- **다중 컬럼 서브쿼리**	여러 컬럼 **조합**이 목록에 포함되는지 비교 (튜플 비교)
다중 컬럼	다중 행	`FROM`	`FROM (...) AS alias`	- 테이블 서브쿼리 - 인라인 뷰 - 파생 테이블	**가상의 테이블**을 생성하여 다른 테이블과 `JOIN` 등 재가공

- 테이블 서브쿼리(파생 테이블)는 표에는 다중 컬럼 다중 행으로 표기했지만 컬럼, 행 수가 단일이어도 쓸 수 있다. 실무에선 통상 다중 컬럼, 다중 행에 자주 사용한다.

서브쿼리 소개

- 서브쿼리는 SQL 쿼리 문 안에 포함된 또 다른 SELECT 쿼리를 의미한다.
- 여러 단계로 나누어 처리해야 할 문제를 하나의 쿼리로 해결할 수 있게 돕는다.
- 서브쿼리가 먼저 실행되고 그 결과가 메인쿼리에서 사용된다.
- 반환하는 행과 열의 수 사용되는 위치에 따라 종류가 나뉜다.

스칼라 서브쿼리

- 단일 행 단일 열의 값을 반환하는 서브쿼리다.
- 스칼라는 단 하나의 값을 의미한다.
- `=`, `>`, `<` 같은 단일 행 비교 연산자와 함께 사용한다.
- 서브쿼리의 결과가 반드시 하나의 행만 반환하도록 주의해야 하며 그렇지 않으면 오류가 발생한다.

다중 행 서브쿼리

- 여러 행의 결과를 반환하는 서브쿼리다.
- `IN`, `ANY`, `ALL` 같은 다중 행 연산자와 함께 사용한다.
- `IN`은 목록에 포함된 값과 일치하는지 확인하며 가장 직관적이고 흔하게 사용된다.
- `ANY`와 `ALL`은 주로 비교 연산자와 쓰이며 `MIN`이나 `MAX` 집계 함수로 대체하는 것이 더 명확할 수 있다.

다중 컬럼 서브쿼리

- 두 개 이상의 컬럼을 반환하는 서브쿼리다.

- `WHERE` 절에서 여러 컬럼을 동시에 비교해야 할 때 유용하다.
- `WHERE (컬럼1, 컬럼2) = (서브쿼리)` 형태로 사용하며 `=` 연산자를 쓸 때는 서브쿼리가 단일 행을 반환해야 한다.
- 서브쿼리가 여러 행을 반환할 때는 `IN` 연산자를 사용한다.

상관 서브쿼리1

- 메인쿼리와 서브쿼리가 서로 연관 관계를 맺고 동작하는 서브쿼리다.
- 메인쿼리의 각 행에 대해 서브쿼리가 반복적으로 실행된다.
- 서브쿼리는 메인쿼리의 컬럼 값을 참조하여 결과를 계산한다.
- 예시로 각 상품이 자신이 속한 카테고리의 평균 가격보다 비싼지 확인할 때 사용한다.

상관 서브쿼리2

- `EXISTS` 연산자는 상관 서브쿼리의 대표적인 활용 사례다.
- `EXISTS`는 서브쿼리의 결과값이 존재하는지 여부만 확인하며 결과 행이 하나라도 있으면 `TRUE`를 반환한다.
- 서브쿼리 테이블이 매우 클 때 `IN` 보다 효율적인 경우가 많다.
- `NOT EXISTS`는 서브쿼리 결과가 존재하지 않을 때 `TRUE`를 반환하여 특정 조건에 해당하지 않는 데이터를 찾을 때 사용한다.

SELECT 서브쿼리

- `SELECT` 절에 위치하는 서브쿼리로 결과가 하나의 컬럼처럼 동작한다.
- 반드시 하나의 값만 반환하는 스칼라 서브쿼리여야 한다.
- 비상관 서브쿼리는 메인쿼리의 모든 행에 동일한 값을 보여준다.
- 상관 서브쿼리는 메인쿼리의 각 행과 상호작용하며 행마다 다른 계산 결과를 보여준다.
- 상관 서브쿼리는 메인쿼리의 행 수만큼 반복 실행되므로 성능 저하에 주의해야 한다.

테이블 서브쿼리

- `FROM` 절에 위치하는 서브쿼리로 인라인 뷰(Inline View)라고도 부른다.
- 서브쿼리의 실행 결과가 하나의 독립된 가상 테이블처럼 사용된다.
- 집계나 그룹핑된 결과를 다시 한번 조인하거나 필터링해야 할 때 유용하다.
- `FROM` 절의 서브쿼리는 반드시 별칭(Alias)을 가져야 한다.

서브쿼리 vs JOIN

- 많은 문제는 서브쿼리와 `JOIN` 두 가지 방법으로 모두 해결할 수 있다.
- 일반적으로 데이터베이스 옵티마이저는 `JOIN`을 더 효율적으로 처리하여 성능이 좋은 경우가 많다.
- 하지만 최신 옵티마이저는 간단한 서브쿼리를 `JOIN`으로 자동 변환하기도 한다.
- 가독성 측면에서는 서브쿼리가 논리적 단계를 명확히 보여줘 더 쉬울 때가 있다.
- `JOIN`을 우선 고려하되 쿼리가 너무 복잡해지면 가독성이 좋은 서브쿼리를 사용하고 성능이 의심될 때는 반드시

측정해야 한다.