7. SQL - 집계와 그룹핑

#0.강의/2.데이터베이스로드맵/1.입문

집계와 그룹핑 실습 데이터 준비

데이터 '집계'와 '그룹핑'의 개념 자체에 집중하기 위해 `order_stat`(주문 통계)라는 이름의 테이블을 만들고, 여기에 분석에 충분한 샘플 데이터를 채워 넣자.

다음 코드를 실행해서 여러분의 데이터베이스에 `order_stat` 테이블을 만들고 데이터를 채워 넣어라.

> **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.

테이블 생성 (CREATE TABLE)

```
CREATE TABLE order_stat (
    order_id INT PRIMARY KEY AUTO_INCREMENT,
    customer_name VARCHAR(50),
    category VARCHAR(50),
    product_name VARCHAR(100),
    price INT,
    quantity INT,
    order_date DATE
);
```

샘플 데이터 삽입 (INSERT INTO)

```
INSERT INTO order_stat (customer_name, category, product_name, price,
quantity, order_date) VALUES
('이순신', '전자기기', '프리미엄 기계식 키보드', 150000, 1, '2025-05-10'),
('세종대왕', '도서', 'SQL 마스터링', 35000, 2, '2025-05-10'),
('신사임당', '가구', '인체공학 사무용 의자', 250000, 1, '2025-05-11'),
('이순신', '전자기기', '고성능 게이밍 마우스', 80000, 1, '2025-05-12'),
('세종대왕', '전자기기', '4K 모니터', 450000, 1, '2025-05-12'),
('장영실', '도서', '파이썬 데이터 분석', 40000, 3, '2025-05-13'),
('이순신', '문구', '고급 만년필 세트', 200000, 1, '2025-05-14'),
('세종대왕', '가구', '높이조절 스탠딩 데스크', 320000, 1, '2025-05-15'),
('신사임당', '전자기기', '노이즈캔슬링 블루투스 이어폰', 180000, 1, '2025-05-15'),
('장영실', '전자기기', '보조배터리 20000mAh', 50000, 2, '2025-05-16'),
('홍길동', NULL, 'USB-C 허브', 65000, 1, '2025-05-17'); -- 카테고리가 NULL인 데이터 추
가
```

실무에서 데이터는 종종 누락, 오류가 섞인다. 때로는 시스템 오류나 사람의 실수로 특정 값이 누락된 채 저장되기도 한다. 물론 정상적으로 의도한 결과일 수 도 있다. 우리는 이런 **'누락된 데이터(NULL)'**가 집계에 어떤 영향을 미치는지 직접 확인해야 한다. 그래서 일부러 마지막에 `category`가 `NULL`인 데이터를 하나 추가했다.

[결과 확인]

```
SELECT * FROM order_stat;
```

order_stat - 데이터

order_id	customer_name	category	product_name	price	quantity	order
1	이순신	전자기기	프리미엄 기계식 키보드	150000	1	2025
2	세종대왕	도서	SQL 마스터링	35000	2	2025
3	신사임당	가구	인체공학 사무용 의자	250000	1	2025

4	이순신	전자기기	고성능 게이밍 마우스	80000	1	2025
5	세종대왕	전자기기	4K 모니터	450000	1	2025
6	장영실	도서	파이썬 데이터 분석	40000	3	2025
7	이순신	문구	고급 만년필 세트	200000	1	2025
8	세종대왕	가구	높이조절 스탠딩 데스크	320000	1	2025
9	신사임당	전자기기	노이즈캔슬링 블루투스 이어폰	180000	1	2025
10	장영실	전자기기	보조배터리 20000mAh	50000	2	2025
11	홍길동	NULL	USB-C 허브	65000	1	2025

> **JOIN**
> 주문 통계(`order_stat`) 테이블은 앞서 설계한 `customers`, `orders`, `products` 테이블을 모두 합쳐(`JOIN`)놓은 것 같아 보인다. 다음 강의에서 학습할 `JOIN`(조인) 기능을 배우면 여러 테이블을 연결해서 사용할 수 있다.

집계 함수

우리 쇼핑몰이 드디어 성공 가도에 올랐다. 매일 수많은 주문이 들어오고, 데이터는 차곡차곡 쌓이고 있다. 이제 우리에게 필요한 것은 무엇일까? 바로 **데이터를 기반으로 한 정확한 의사결정**이다.

대표가 이렇게 묻는다. "그래서, 지금까지 총 몇 건이나 주문이 들어왔어? 총매출은 얼마야? 우리 쇼핑몰에서 가장 비싸게 팔린 상품은 뭐고, 가장 싸게 팔린 상품은 뭐지? 그리고 지금까지 우리 쇼핑몰에서 구매한 순수 고객 수는 몇 명이야?"

> 실무에서는 진짜로 이런 다양한 요청이 들어온다.

이 질문에 답하기 위해 `order_stat` 테이블의 데이터를 하나씩 눈으로 세고, 계산기로 더하고 있을 것인가? 데이터가 수만, 수백만 건이 된다면 불가능한 일이다. 바로 이런 문제를 해결하기 위해 SQL은 강력하고 편리한 **집계 함수**(Aggregate Functions)를 제공한다. 집계 함수란 여러 행의 데이터를 바탕으로 하나의 요약된 결과 값을 계산해주는 함수를 말한다.

집계 함수 (Aggregate Functions)
여러 행의 값을 바탕으로 단일 결과 값을 계산한다. `GROUP BY` 절과 함께 자주 사용된다.

함수	설명
`AVG(expression)`	표현식의 평균값을 반환한다.
`COUNT(expression)`	표현식의 결과가 `NULL` 이 아닌 행의 수를 반환한다.
`COUNT(*)`	테이블의 전체 행 수를 반환한다.
`MAX(expression)`	표현식의 최댓값을 반환한다.
`MIN(expression)`	표현식의 최솟값을 반환한다.
`SUM(expression)`	표현식의 합계를 반환한다.

전체 데이터 건수 파악하기: `COUNT()`

가장 기본적인 질문부터 해결해 보자. "우리 쇼핑몰의 총 주문 건수는 몇 건일까?"

`COUNT()` 함수는 특정 컬럼의 행(row) 개수를 세는 가장 기본적인 집계 함수다. `COUNT(*)`는 **NULL 값에 상관없이 테이블의 모든 행의 개수**를 센다.

예제: 총 주문 건수 파악하기
가장 먼저, 우리 쇼핑몰에 들어온 총 주문 건수를 파악해 보자. 우리가 만든 `order_stat` 테이블의 전체 행 개수를 세면 된다.

```sql
SELECT COUNT(*) FROM order_stat;
```

실행 결과

COUNT(*)
11

`order_stat` 테이블에는 이제 총 11개의 행이 있으므로, 총 주문 건수는 11건이라는 것을 알 수 있다.
`COUNT(*)` 는 이처럼 테이블의 전체 규모를 파악하는 데 가장 기본적으로 사용된다.

NULL과 특정 컬럼의 개수: `COUNT(컬럼)` vs `COUNT(*)`

방금 우리는 `COUNT(*)` 로 전체 주문이 11건이라는 것을 알아냈다. 그런데 여기서 또 다른 질문이 생긴다. "그러면, **카테고리가 등록된 주문**은 총 몇 건이지?"

우리가 추가한 11번째 데이터는 `category` 값이 `NULL` 이다. 즉, 카테고리 정보가 누락되었다. 이런 상황에서 전체 개수가 아닌, 특정 컬럼에 값이 실제로 존재하는 데이터의 개수만 세려면 어떻게 해야 할까? `COUNT(*)` 만으로는 이 문제를 해결할 수 없다. 바로 이 차이 때문에 `COUNT` 와 같은 집계 함수는 조금 더 깊이 이해해야 한다.

개념 비교: `COUNT(*)` vs `COUNT(컬럼명)`

- `COUNT(*)` : 별표(`*`)는 '모든 컬럼'을 의미하며, 더 나아가 그냥 **'행 자체'**를 가리킨다. 따라서 `COUNT(*)` 는 `NULL` 값의 존재 여부와 상관없이 **테이블의 물리적인 행의 개수**를 모두 센다. 마치 출석부의 총인원을 세는 것과 같다.
- `COUNT(컬럼명)` : 특정 컬럼 이름을 지정하면, 해당 컬럼의 값 중에서 `NULL` **이 아닌 값의 개수만 센다.** `NULL` 은 '값이 없음'을 의미하므로 개수에서 제외되는 것이다. 이것은 출석부에서 이름이 실제로 적힌 학생 수만 세는 것과 같다.

예제: `COUNT(*)` 와 `COUNT(category)` 의 차이 확인하기
이제 실제 쿼리를 통해 `COUNT(*)` 와 `COUNT(category)` 의 차이를 눈으로 직접 확인해 보자.

```
SELECT
    COUNT(*) AS `전체 주문 건수`,
    COUNT(category) AS `카테고리 등록 건수`
FROM
    order_stat;
```

✨ 별칭에 공백 같은 특수 문자가 필요하다면 백틱(`)으로 감싸라

실행 결과

전체 주문 건수	카테고리 등록 건수
11	10

결과가 모든 것을 명확하게 설명해 준다.

- `COUNT(*)` : `NULL`을 포함한 모든 행을 세어서 총 **11**건을 반환했다.
- `COUNT(category)` : `category` 컬럼에서 값이 `NULL`인 '홍길동'의 주문을 제외하고, 값이 존재하는 행만 세어서 **10**건을 반환한다.

합계와 평균 계산으로 매출 분석하기: `SUM()`, `AVG()`

"지금까지의 총매출액은 얼마인가?" 또는 "고객들은 한 번 주문할 때 평균적으로 얼마를 쓰는가?" 와 같은 질문은 쇼핑몰의 수익성을 파악하는 데 핵심적이다. 이런 계산을 위해 `SUM()`과 `AVG()` 함수를 사용한다.

- `SUM(컬럼명)` : 지정한 숫자 컬럼의 모든 값을 더하여 **합계**를 계산한다.
- `AVG(컬럼명)` : 지정한 숫자 컬럼의 값들의 **평균**을 계산한다.

`SUM()`과 `AVG()` 함수는 계산 과정에서 **NULL 값을 자동으로 제외**한다. `NULL`을 `0`으로 취급하지 않는다는 점을 명심하자.

예제 1: 총 매출액과 평균 주문 금액 분석하기

`order_stat` 테이블을 사용해서 우리 쇼핑몰의 총매출액과, 주문 건당 평균 구매액(객단가)을 구해보자. 총매출액은 각 주문의 '상품 가격(`price`)'과 '주문 수량(`quantity`)'을 곱한 값의 합계로 계산해야 한다.

```
SELECT
    SUM(price * quantity) AS `총 매출액`,
    AVG(price * quantity) AS `평균 주문 금액`
FROM
    order_stat;
```

실행 결과

총 매출액	평균 주문 금액
1985000	180454.5455

이 쿼리 한 줄로 우리 쇼핑몰의 총매출이 1,985,000원이고, 주문 한 건당 평균적으로 약 180,455원을 지출한다는 결과를 얻었다.

예제 2: 총 판매 상품 수량과 주문당 평균 수량 분석하기
이번에는 고객들이 총 몇 개의 상품을 주문했고, 한 번 주문할 때 평균적으로 몇 개를 주문하는지 분석해 보자. `quantity` 컬럼을 사용하면 된다.

```
SELECT
    SUM(quantity) AS `총 판매 수량`,
    AVG(quantity) AS `주문당 평균 수량`
FROM
    order_stat;
```

실행 결과

총 판매 수량	주문당 평균 수량
15	1.3636

지금까지 총 15개의 상품이 판매되었고, 주문 한 건당 평균 1.3636개의 상품을 구매했다는 사실을 알 수 있다. 이처럼 `SUM`과 `AVG`를 활용하면 데이터의 전체적인 규모와 평균적인 경향을 손쉽게 파악할 수 있다.

최대, 최소값으로 상품 전략 세우기: `MAX()`, `MIN()`

"우리 쇼핑몰에서 가장 비싸게 팔린 단일 상품의 가격은 얼마인가?" 또는 "가장 오래된 주문은 언제였는가?" 와 같은 정보는 가격 정책을 수립하거나 고객 활동 기간을 파악하는 데 중요한 단서가 된다. `MAX()`와 `MIN()` 함수를 사용하면 이런 정보를 즉시 얻을 수 있다.

- `MAX(컬럼명)` : 지정한 컬럼에서 **최댓값**을 찾는다.

- `MIN(컬럼명)` : 지정한 컬럼에서 **최솟값**을 찾는다.

예제 1: 최고가, 최저가 상품 가격 찾기

`order_stat` 테이블에서 판매된 상품 중 가장 비싼 상품의 가격과 가장 저렴한 상품의 가격을 찾아보자.

```
SELECT
    MAX(price) AS 최고가,
    MIN(price) AS 최저가
FROM
    order_stat;
```

실행 결과

최고가	최저가
450000	35000

가장 비싼 상품은 450,000원, 가장 저렴한 상품은 35,000원이라는 것을 바로 알 수 있다. 이를 통해 우리 쇼핑몰의 가격대가 어떻게 형성되어 있는지 파악하고, 향후 상품 기획에 참고할 수 있다.

예제 2: 최초 주문일과 최근 주문일 찾기

`order_stat` 테이블에서 가장 먼저 들어온 주문의 날짜와 가장 최근에 들어온 주문의 날짜를 찾아보자.
`order_date` 컬럼에 `MIN()`과 `MAX()`를 적용하면 된다.

```
SELECT
    MIN(order_date) AS `최초 주문일`,
    MAX(order_date) AS `최근 주문일`
FROM
    order_stat;
```

실행 결과

최초 주문일	최근 주문일
2025-05-10	2025-05-17

우리 쇼핑몰의 첫 주문은 2025년 5월 10일에, 가장 최근 주문은 2025년 5월 17일에 들어왔다는 것을 알 수 있다. 이처럼 `MAX`와 `MIN`은 데이터의 범위를 파악하는 데 매우 유용하다.

고유 고객 수 파악하기: `DISTINCT`

"지금까지 우리 쇼핑몰을 통해 주문한 고객은 총 몇 명일까?"

이 질문에 답하기 위해 `COUNT(customer_name)`을 사용하면 될까? 그렇게 하면 총 주문 건수인 11이 나올 것이다. 왜냐하면 '이순신' 고객이나 '세종대왕' 고객처럼 여러 번 주문한 고객의 이름이 중복해서 세기 때문이다.

이처럼 중복된 값을 제외하고 **순수한(고유한) 값의 개수**만 세고 싶을 때 사용하는 것이 바로 `DISTINCT` 키워드다.

`DISTINCT`는 집계 함수와 함께 `COUNT(DISTINCT 컬럼명)` 형태로 사용할 수 있다. 이때는 특정 컬럼에서 중복을 제거한 뒤 개수를 센다.

예제: 실제 구매자 수 정확히 파악하기

`order_stat` 테이블에서 중복을 포함한 고객 이름의 수와, 중복을 제거한 순수 고객 수를 함께 조회하여 그 차이를 명확하게 비교해 보자.

```
SELECT
    COUNT(customer_name) AS `총 주문 건수`,
    COUNT(DISTINCT customer_name) AS `순수 고객 수`
FROM
    order_stat;
```

실행 결과

총 주문 건수	순수 고객 수
11	5

결과가 모든 것을 말해준다.

- `COUNT(customer_name)`: 중복을 포함하여 총 주문 건수인 11을 반환했다.

- `COUNT(DISTINCT customer_name)` : 고객 이름('이순신', '세종대왕', '신사임당', '장영실', '홍길동')에서 중복을 모두 제거하고 고유한 값들의 개수인 5를 반환했다.

따라서 우리 쇼핑몰에서 한 번이라도 구매한 이력이 있는 **순수 고객의 수는 5명**이라는 것을 정확하게 파악할 수 있다.

GROUP BY - 그룹으로 묶기

이제 우리는 쇼핑몰의 전체 주문 건수, 총매출 등을 파악할 수 있게 되었다. 하지만 열정적인 대표의 질문은 여기서 멈추지 않는다.

- "어떤 상품 카테고리가 가장 인기가 많을까? **카테고리별 주문 건수**를 알려줘, 그리고 **카테고리별로 매출액이 얼마나 되는지** 알려줘."
- "전체 실적은 알겠는데, **어떤 고객이 우리 쇼핑몰의 진짜 VIP지?** 즉, 고객별로 총 얼마를 썼고 몇 개의 상품을 구매했는지 순서대로 보고 싶어."

이런 질문들은 `SUM()`이나 `COUNT()` 같은 집계 함수만으로는 해결할 수 없다. 왜냐하면 '전체'가 아닌 **'상품 카테고리별로'**, **'고객 별로'** 묶어서 집계해야 하기 때문이다. 전체 데이터를 특정 기준에 따라 의미 있는 여러 그룹으로 나눈 뒤, 각 그룹에 대해 요약 통계를 내야 하는 상황이다.
쉽게 이야기해서 학교에 1학년, 2학년 3학년이 있다면, 학교 전체의 평균 점수가 아니라 1학년의 평균 점수, 2학년의 평균, 3학년의 평균 점수를 각각 구해야 하는 경우이다.

바로 이럴 때 사용하는 것이 `GROUP BY` 절이다.

그룹화의 첫걸음: `GROUP BY` 기본

`GROUP BY`는 이름 그대로, 특정 컬럼의 값이 같은 행들을 하나의 그룹으로 묶어주는 역할을 한다.

개념 설명: `GROUP BY` 절에 그룹화의 기준이 될 컬럼을 지정한다. 이렇게 그룹화된 결과에 집계 함수를 적용하면, 전체에 대한 통계가 아닌 '각 그룹에 대한 통계'를 낼 수 있다.

예제: 카테고리별 주문 건수
첫 번째 질문, "어떤 상품 카테고리가 가장 인기가 많을까?"에 답하기 위한 첫 단계로, **'카테고리별로 주문이 몇 건씩 발생했는지'** 먼저 세어보자.

`order_stat` 테이블의 데이터를 `category` 컬럼을 기준으로 그룹으로 묶는다. '전자기기'는 '전자기기'끼리, '도서'는 '도서'끼리 묶는 것이다. 그런 다음 각 그룹에 `COUNT(*)` 를 적용하면, 각 카테고리 그룹에 속한 주문(행)의 개수를 셀 수 있다.

```
SELECT
    category,
    COUNT(*) AS `카테고리별 주문 건수`
FROM
    order_stat
GROUP BY
    category;
```

실행 결과

category	카테고리별 주문 건수
전자기기	5
도서	2
가구	2
문구	1
NULL	1

결과 분석

결과를 보자. '전자기기' 카테고리에서 5건의 주문이 발생하여 가장 인기가 많다는 것을 한눈에 알 수 있다. `GROUP BY category` 는 전체 데이터를 카테고리 값에 따라 그룹으로 나누고, `COUNT(*)` 는 각 그룹에 몇 개의 주문이 있는지 계산해 준다.

여기서 주목할 점은 `category` 가 `NULL` 인 데이터도 하나의 그룹으로 묶여서 결과에 포함되었다는 것이다. `GROUP BY` 는 `NULL` 값 또한 하나의 독립된 그룹으로 취급하여 집계한다. 실무에서는 이런 `NULL` 그룹을 통해 데이터가 누락되었음을 발견하고 데이터 정제의 필요성을 인지할 수 있다.

예제: 고객별로 총 몇 번이나 주문했을까?

`order_stat` 테이블에는 한 고객이 여러 번 주문한 기록이 섞여 있다. `GROUP BY` 를 사용해서 `customer_name`

이 같은 행들을 하나의 그룹으로 묶어보자. 그리고 각 그룹에 대해 `COUNT(*)`를 실행하면 '고객별 주문 횟수'를 구할 수 있다.

```
SELECT
    customer_name,
    COUNT(*) AS `주문 횟수`
FROM
    order_stat
GROUP BY
    customer_name;
```

실행 결과

customer_name	주문 횟수
이순신	3
세종대왕	3
신사임당	2
장영실	2
홍길동	1

결과를 보자. `GROUP BY customer_name`은 `order_stat` 테이블의 11개 행을 고객 이름에 따라 5개의 그룹으로 묶었다. 그리고 `SELECT` 절의 `COUNT(*)`는 각 그룹에 속한 행의 개수를 세어 반환했다. 이로써 우리는 각 고객이 몇 번씩 주문했는지 명확하게 알 수 있다.

그룹별로 심층 분석하기: `GROUP BY`와 집계 함수

`GROUP BY`의 진정한 강력함은 다양한 집계 함수와 함께 사용될 때 발휘된다. `COUNT()`뿐만 아니라 `SUM()`, `AVG()`, `MAX()`, `MIN()` 등을 모두 적용할 수 있다.

예제 시나리오: 고객별 구매 활동 분석 (VIP 고객 찾기)

이번에는 한 걸음 더 나아가, 고객별로 총 주문 횟수, 총 구매 수량, 그리고 총 구매 금액을 함께 계산해 보자. 이를 통해 어떤 고객이 우리 쇼핑몰의 진정한 VIP인지 찾아낼 수 있다.

```sql
SELECT
    customer_name,
    COUNT(*) AS `총 주문 횟수`,
    SUM(quantity) AS `총 구매 수량`,
    SUM(price * quantity) AS `총 구매 금액`
FROM
    order_stat
GROUP BY
    customer_name
ORDER BY
    `총 구매 금액` DESC; -- 백틱 사용 주의!
```

- `ORDER BY`를 추가하여 총 구매 금액이 높은 순으로 정렬하면 VIP 고객을 한눈에 파악하기 좋다.

실행 결과

customer_name	총 주문 횟수	총 구매 수량	총 구매 금액
세종대왕	3	4	840000
이순신	3	3	430000
신사임당	2	2	430000
장영실	2	5	220000
홍길동	1	1	65000

이 결과를 보면 세종대왕이 총 840,000원을 사용하여 우리 쇼핑몰의 최고 VIP 고객임을 명확히 알 수 있다. 이처럼 `GROUP BY`와 집계 함수를 조합하면 강력한 비즈니스 분석이 가능하다.

주의! - ORDER BY 백틱 사용

- `ORDER BY`에서 `총 구매 금액` 문자에 백틱(`)을 사용했다. 이렇게 백틱을 사용해야 SELECT에서 사용한 컬럼명으로 인식한다.
- MySQL에서 작은 따옴표('), 큰 따옴표(")를 사용하면 문자 상수로 인식한다. 따라서 SELECT에서 사용한 컬럼명으로 인식하지 못한다.
    - 이렇게 되면 모두 같은 문자 상수로 정렬되어 버린다. 여기서는 모든 컬럼이 "총 구매 금액"이라는 문자 값으로 정렬되므로 정렬 효과가 없다. 따라서 정렬이 안되는 문제가 있다.

작은 따옴표를 사용하는 잘못된 예시

```
SELECT
    customer_name,
    COUNT(*) AS `총 주문 횟수`,
    SUM(quantity) AS `총 구매 수량`,
    SUM(price * quantity) AS `총 구매 금액`,
    '총 구매 금액' AS `정렬 값`
FROM
    order_stat
GROUP BY
    customer_name
ORDER BY
    '총 구매 금액' DESC; -- 작은 따옴표('), 문자로 인식
```

- 이렇게 되면 모든 컬럼이 '총 구매 금액'이라는 똑같은 문자 값으로 정렬된다. 따라서 정렬이 정상적으로 이루어지지 않는다.

[실행 결과]

customer_name	총 주문 횟수	총 구매 수량	총 구매 금액	정렬 값
이순신	3	3	430000	총 구매 금액
세종대왕	3	4	840000	총 구매 금액
신사임당	2	2	430000	총 구매 금액
장영실	2	5	220000	총 구매 금액
홍길동	1	1	65000	총 구매 금액

- 이해를 돕기 위해 정렬 값 필드를 추가했다.
- 모두 똑같은 '총 구매 금액'이라는 문자를 기준으로 정렬된다.
- 결과적으로 총 구매 금액으로 정렬되지 않았다.

더 세분화된 그룹으로 분석하기: 여러 컬럼 기준 그룹화

분석은 때로 더 깊이 들어가야 한다.

대표님의 새로운 요구사항이 등장한다. "고객별 분석은 알겠는데, 그럼 **어떤 고객이 어떤 카테고리의 상품을** 주로 구매

하는지 보고 싶다."
이런 요구사항은 어떻게 해결할까?

`GROUP BY` 절에 여러 컬럼을 콤마(`,`)로 구분하여 나열하면, 다중 기준으로 데이터를 그룹화할 수 있다. `GROUP BY 컬럼1, 컬럼2`는 컬럼1로 먼저 그룹화하고, 그 안에서 다시 컬럼2로 그룹화하여 더 세분화된 그룹을 만든다.

예제 시나리오: 고객이 어떤 카테고리에서 얼마를 사용했는가?
'어떤 고객이(`customer_name`)', '어떤 상품 카테고리에서(`category`)' 얼마를 썼는지 그룹으로 묶어보자.

```
SELECT
    customer_name,
    category,
    SUM(price * quantity) AS `카테고리별 구매 금액`
FROM
    order_stat
GROUP BY
    customer_name, category
ORDER BY
    customer_name, `카테고리별 구매 금액` DESC;
```

실행 결과

customer_name	category	카테고리별 구매 금액
세종대왕	전자기기	450000
세종대왕	가구	320000
세종대왕	도서	70000
신사임당	가구	250000
신사임당	전자기기	180000
이순신	전자기기	230000
이순신	문구	200000
장영실	도서	120000
장영실	전자기기	100000

홍길동	NULL	65000

이 결과는 매우 구체적인 정보를 제공한다. 예를 들어 세종대왕이 전자기기 카테고리에서 구매한 금액의 합, 가구에서 구매한 금액의 합, 도서에서 구매한 합을 각각 따로 구할 수 있다.

GROUP BY - 주의사항

`GROUP BY`를 사용할 때 `SELECT` 절에는 `GROUP BY`**에 사용된 컬럼**과 **집계 함수**만 사용할 수 있다.

왜 그럴까? 상식적으로 생각해 보자. 우리가 데이터를 어떤 기준으로 그룹화했는데, 그 그룹에 속한 개별 데이터 중 하나를 불쑥 달라고 하면 데이터베이스는 어떤 것을 줘야 할까? 그룹에는 여러 데이터가 섞여 있으니, 그중 하나의 값을 특정할 수 없기 때문이다. 데이터베이스는 이렇게 모호한 요청을 허용하지 않는다. 따라서 그룹을 대표하는 값(`GROUP BY` 컬럼)이나, 그룹 전체를 요약한 값(집계 함수)만 조회할 수 있도록 규칙을 정한 것이다.

다음 예시를 통해 자세히 알아보자.

어떤 상품명을 보여줘야 할까?

"카테고리별로 묶어서 상품 개수를 보고 싶다"는 요구 사항을 받았다고 가정해보자.
다음과 같이 아주 간단하게 문제를 해결할 수 있다.

```
SELECT
    category,
    COUNT(*)
FROM
    order_stat
GROUP BY
    category;
```

[실행 결과]

category	COUNT(*)
전자기기	5

도서	2
가구	2
문구	1
NULL	1

그런데 이 요구 사항에 추가로 "각 카테고리에 속한 상품명도 하나 보고 싶다"는 요구를 받았다고 가정해보자.
아주 자연스러운 요구처럼 들린다. 그래서 다음과 같이 `product_name`을 추가로 조회하는 쿼리를 작성했다고 가정해보자.

```
-- 잘못된 쿼리의 예시
SELECT
    category,
    product_name, -- 바로 이 컬럼이 문제다!
    COUNT(*)
FROM
    order_stat
GROUP BY
    category;
```

[실행 결과]

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

- `product_name`이 `GROUP BY` 절에 없다는 오류이다.

이 쿼리는 대부분의 데이터베이스에서 오류를 발생시킨다. 왜일까? 데이터베이스의 입장이 되어 직접 데이터를 살펴보자.

'전자기기' 카테고리 그룹의 내부

`GROUP BY category` 명령을 내리면, 전자기기, 도서, 가구, 문구, NULL 각각의 카테고리를 그룹으로 묶는다.

여기서 '전자기기' 카테고리의 데이터만 자세히 살펴보자.

order_id	customer_name	category	product_name	price	quantity	order
1	이순신	전자기기	프리미엄 기계식 키보드	150000	1	2025
4	이순신	전자기기	고성능 게이밍 마우스	80000	1	2025
5	세종대왕	전자기기	4K 모니터	450000	1	2025
9	신사임당	전자기기	노이즈캔슬링 블루투스 이어폰	180000	1	2025
10	장영실	전자기기	보조배터리 20000mAh	50000	2	2025

데이터베이스는 '전자기기' 카테고리에 속한 주문 5건을 하나의 그룹으로 묶어야 한다. 쉽게 이야기해서 5행의 데이터를 **단 한 행**으로 만들어야 한다.

SELECT에서 `category`, `product_name`, `COUNT(*)`를 선택했기 때문에 각각의 대표를 하나만 선택해야 한다.

- `category`는 무엇인가? 그룹의 기준이니 당연히 '전자기기'이다. '전자기기'라는 하나의 명확한 값을 구할 수 있다.
  - `category` 전자기기 그룹에 속한 모든 값은 같은 '전자기기'이다.
- `COUNT(*)`는 무엇인가? 그룹 내 주문이 5건이니, 5라는 명확한 하나의 값을 구할 수 있다.
- **그런데 `product_name`은 무엇을 보여줘야 할까?** '프리미엄 기계식 키보드', '고성능 게이밍 마우스', '4K 모니터', '노이즈캔슬링 블루투스 이어폰', '보조배터리 20000mAh' 이 다섯 가지 값 중에 어떤 것을 선택해야 하는가? 첫 번째 값? 마지막 값? 아니면 가장 비싼 상품의 이름? 데이터베이스는 이 질문에 대한 답을 알 수 없다. 요청 자체가 모호하기 때문이다.

이것은 마치 학교 선생님에게 "3반의 학생 수와, 3반의 학생 이름도 한 명 알려주세요"라고 하는 것과 같다. '학생 수'는 반 전체를 대표하는 명확한 하나의 값이지만, 3반의 '학생 이름'은 30명이나 되기 때문에 누구의 이름을 말해야 할지 알 수 없는 것과 같은 이치다.

올바른 해결책

데이터베이스는 이처럼 모호한 명령을 처리할 수 없도록 설계되었다. 그래서 "GROUP BY로 묶었다면, SELECT 절에

는 그룹의 '대표 선수'인 `GROUP BY` 컬럼이나, 그룹의 '요약 정보'인 집계 함수만 올 수 있다"는 엄격한 규칙을 적용한다.

따라서 '전자기기' 그룹에서는 그룹을 대표하는 `category`인 '전자기기'와, 그룹 전체를 요약한 `COUNT(*)`, `SUM(price * quantity)` 등과 같은 값만 함께 조회할 수 있다. 그룹 전체를 대표할 수 없는 개별 행의 정보(`product_name`, `order_id`, `customer_name` 등)는 `SELECT` 절에 단독으로 올 수 없다. 이 점을 반드시 기억해야 실무에서 `GROUP BY`를 사용할 때 발생하는 흔한 오류를 피할 수 있다.

정리하면 GROUP BY를 사용하는 경우 GROUP BY에 사용된 컬럼이나 집계 함수만 사용할 수 있다.

```
SELECT
    category, -- GROUP BY에 사용된 컬럼
    product_name, -- GROUP BY에 사용된 컬럼이 아님X
    quantity, -- GROUP BY에 사용된 컬럼이 아님X
    COUNT(*), -- 집계 함수
    SUM(quantity) -- 집계 함수
FROM
    order_stat
GROUP BY
    category;
```

- `product_name`, `quantity`는 GROUP BY에 사용된 컬럼이 아니고, 집계 함수를 사용한 것도 아니므로 오류가 발생한다.

HAVING - 그룹 필터링1

`GROUP BY`를 통해 우리는 다양한 기준으로 데이터를 묶고 집계하는 법을 배웠다. 이제 회사의 대표가 새로운 질문을 던진다.
"우리 쇼핑몰의 카테고리별 실적은 알겠는데, 이 중에서 **매출액이 50만 원을 넘는 핵심 카테고리만** 따로 보고 싶어."

우리는 먼저 `GROUP BY`를 사용해서 각 카테고리별로 총 얼마의 매출이 발생했는지 계산해야 한다.

```
SELECT
    category,
    SUM(price * quantity) AS total_sales
FROM
```

```
    order_stat
GROUP BY
    category;
```

이 쿼리는 `order_stat` 테이블의 모든 주문 기록을 `category` 기준으로 묶고, 각 그룹(카테고리)별로 `price * quantity`의 합계를 계산한다.

[실행 결과]

category	total_sales
전자기기	960000
도서	190000
가구	570000
문구	200000
NULL	65000

이런 요구사항을 우리가 이미 배운 `WHERE` 절로 해결할 수 있을까? `WHERE SUM(price * quantity) >= 500000` 와 같이 말이다.

```
SELECT
    category,
    SUM(price * quantity) AS total_sales
FROM
    order_stat
WHERE
    SUM(price * quantity) >= 500000
GROUP BY
    category;
```

실행 결과

```
Error Code: 1111. Invalid use of group function
```

- 오류가 발생한다.

왜 그럴까? 이유는 `WHERE` **절은 그룹화가 이루어지기 전, 즉 테이블의 개별 행 하나하나에 대해 조건을 검사**하기 때문이다. `SUM()`과 같은 집계 함수는 `GROUP BY`를 통해 여러 행이 하나의 그룹으로 묶인 후에야 계산될 수 있는 값이다. 따라서 `WHERE` 절의 입장에서는 아직 존재하지도 않는 값을 조건으로 사용하려는 셈이니, 오류를 내뱉는 것이 당연하다.

SQL 작동 순서: `FROM` ➜ `WHERE` ➜ `GROUP BY` ➜ `...` ➜ `SELECT` ➜ `ORDER BY`

바로 이처럼, **그룹으로 묶은 결과에 대해 다시 필터링**을 하고 싶을 때, SQL은 `HAVING`이라는 도구를 제공한다. `WHERE`가 개인전의 예선이라면, `HAVING`은 팀 대항전의 본선이라고 할 수 있다.

WHERE, HAVING 개념 비유

1. 전체 학생(테이블 데이터)들이 운동장에 모여있다.
2. 선생님이 "안경 쓴 학생들만 앞으로 나와!"라고 외친다. (`WHERE`: 개별 행 필터링)
3. 앞으로 나온 학생들을 대상으로 "같은 반 학생들끼리 모여!"라고 한다. (`GROUP BY`: 그룹화)
4. 반별로 모인 그룹들을 보며 선생님이 마지막으로 "이 중에서, 반 평균 점수가 80점 이상인 그룹만 남아!"라고 외친다. (`HAVING`: 그룹 필터링)

`WHERE`는 그룹화 이전에 개개인을 걸러내는 조건이고, `HAVING`은 그룹화 이후에 그룹 자체를 걸러내는 조건인 것이다.

그룹을 위한 필터: `HAVING` 절의 사용법

`HAVING` 절은 `GROUP BY` 절 바로 뒤에 위치하며, 그룹화된 결과에 대한 조건을 지정하는 역할을 한다. `WHERE`가 원본 데이터의 개별 행을 위한 필터라면, `HAVING`은 `GROUP BY`를 통과한 그룹을 위한 필터다.

- **개념 설명**: `HAVING` 절의 조건문에는 `SUM()`, `COUNT()`, `AVG()` 같은 집계 함수를 사용할 수 있다. `GROUP BY`를 통해 생성된 여러 그룹들 중에서 `HAVING` 절의 조건을 만족하는 그룹들만 최종 결과에 포함된다.
- **작동 순서**: `FROM` ➜ `WHERE` ➜ `GROUP BY` ➜ `HAVING` ➜ `SELECT` ➜ `ORDER BY` 순서로 동작한다는 것을 반드시 기억해야 한다. `GROUP BY`로 그룹을 만들고, `HAVING`으로 그 그룹들을 걸러낸다.

예제 1: '핵심 카테고리' 필터링하기

대표의 요구사항, "총 매출액이 50만 원 이상인 핵심 카테고리"를 찾아보자. 이 과정을 단계별로 나누어 `HAVING`의 역할을 명확히 이해해 보자.

1단계: 먼저 카테고리별로 그룹화하여 총 매출액 집계하기

`HAVING`을 사용하기 전에, 우리는 먼저 `GROUP BY`를 사용해서 각 카테고리별로 총 얼마의 매출이 발생했는지 계산해야 한다.

```
SELECT
    category,
    SUM(price * quantity) AS total_sales
FROM
    order_stat
GROUP BY
    category;
```

이 쿼리는 `order_stat` 테이블의 모든 주문 기록을 `category` 기준으로 묶고, 각 그룹(카테고리)별로 `price * quantity`의 합계를 계산한다.

[1단계 실행 결과]

category	total_sales
전자기기	960000
도서	190000
가구	570000
문구	200000
NULL	65000

결과를 보면 각 카테고리의 총 매출액이 잘 계산되었다. 이제 이 결과 중에서 '총 매출액 50만 원 이상'이라는 조건을 만족하는 그룹만 걸러내면 된다. 바로 이럴 때 `HAVING`이 등장한다.

2단계: `HAVING` 절로 2차 필터링

이제 1단계 쿼리에 `HAVING` 절을 추가해서 '총 매출액이 50만 원 이상'인 그룹만 남겨보자.

```
SELECT
    category,
    SUM(price * quantity) AS total_sales
FROM
```

```
    order_stat
GROUP BY
    category
HAVING
    SUM(price * quantity) >= 500000;
```

[2단계 실행 결과]

category	total_sales
전자기기	960000
가구	570000

`GROUP BY`의 결과로 나왔던 5개의 카테고리 그룹 중에서, 총 매출액이 50만 원 미만이었던 '도서', '문구', 그리고 `NULL` 카테고리는 `HAVING` 절의 `SUM(...) >= 500000` 조건을 만족하지 못해 필터링되어 제외되었다. 이처럼 `HAVING`은 `GROUP BY`의 결과에 대한 필터링 조건을 적용한다.

HAVING에서 별칭 사용

`HAVING`에 `SUM(price * quantity)` 대신에 SELECT 절에서 사용한 `total_sales` 별칭을 대신 사용해보자.

```
SELECT
    category,
    SUM(price * quantity) AS total_sales
FROM
    order_stat
GROUP BY
    category
HAVING
    total_sales >= 500000;
```

- MySQL에서는 잘 작동한다.

SQL 표준과 HAVING에서 별칭 사용

- **SQL 표준:** 표준 SQL 문법에 따르면 `HAVING` 절은 `SELECT` 절보다 먼저 처리된다. 따라서 `SELECT` 절에서 지정한 별칭(`alias`)을 `HAVING` 절에서 사용하는 것은 원칙적으로 불가능하다.

- **DBMS별 차이:**
  - **MySQL, PostgreSQL 등:** 사용자의 편의를 위해 표준을 확장하여 `HAVING` 절에서 `SELECT`의 별칭을 사용할 수 있도록 허용한다.
  - **SQL Server, Oracle 등:** SQL 표준을 비교적 엄격하게 준수하므로 `HAVING` 절에서 별칭을 사용하면 오류가 발생한다.
- **결론:** 여러 데이터베이스 환경에서 코드가 문제없이 작동하도록 하려면, 설명한 대로 `HAVING` 절에 `SUM(price * quantity)`와 같이 집계 함수 표현식을 직접 사용하는 것이 **안전하고 호환성이 높은 방법**이다.

예제 2: 충성 고객 필터링하기

이번에는 다른 기준으로 그룹을 필터링해보자. '3회 이상 주문한' 충성 고객을 찾아보자. 이 또한 2단계로 나누어 생각하면 아주 간단하다.

1단계: 고객별 총 주문 횟수 집계하기

먼저 각 고객이 몇 번이나 주문했는지 `COUNT(*)`를 사용해서 계산해 보자.

```
SELECT
    customer_name,
    COUNT(*) AS order_count
FROM
    order_stat
GROUP BY
    customer_name;
```

[1단계 실행 결과]

customer_name	order_count
이순신	3
세종대왕	3
신사임당	2
장영실	2
홍길동	1

고객별로 총 몇 번의 주문(행)이 있었는지 정확히 집계되었다. 이제 여기서 주문 횟수가 3회 이상인 고객만 남기면 된다.

2단계: `HAVING` 절을 추가하여 주문 횟수 3회 이상인 그룹 필터링하기

1단계 결과에 `HAVING COUNT(*) >= 3` 이라는 조건을 추가하여 우리가 원하는 충성 고객 목록을 완성해 보자.

```
SELECT
    customer_name,
    COUNT(*) AS order_count
FROM
    order_stat
GROUP BY
    customer_name
HAVING
    COUNT(*) >= 3;
```

[2단계 실행 결과]

customer_name	order_count
이순신	3
세종대왕	3

총 주문 횟수가 3회 미만이었던 신사임당, 장영실, 홍길동 그룹은 `HAVING` 절의 조건을 통과하지 못하고, 3회씩 주문한 이순신과 세종대왕만이 최종 결과에 포함되었다. 이처럼 `HAVING`을 사용하면 그룹화된 데이터에 대해 원하는 조건으로 정밀하게 필터링할 수 있다.

HAVING - 그룹 필터링2

`WHERE` vs `HAVING`: 무엇이, 언제 다른가?

SQL을 처음 다룰 때 가장 헷갈리는 부분 중 하나가 바로 `WHERE`와 `HAVING`의 차이점이다. 둘 다 '필터링'을 하지만,

작동하는 시점과 대상이 완전히 다르다는 것을 이제 명확히 구분해야 한다.

구분	`WHERE`	`HAVING`
작동 시점	`GROUP BY` 이전	`GROUP BY` 이후
필터링 대상	개별 행 (Row)	그룹 (Group)
사용 가능 함수	집계 함수 사용 불가	집계 함수 사용 가능
역할	개별 행을 선택	그룹화된 결과 중에서 조건에 맞는 그룹만 선택

이해를 돕기 위해 앞서 사용한 비유를 다시 사용하겠다.

1. 전체 학생(테이블 데이터)들이 운동장에 모여있다.
2. 선생님이 "안경 쓴 학생들만 앞으로 나와!"라고 외친다. (`WHERE`: 개별 행 필터링)
3. 앞으로 나온 학생들을 대상으로 "같은 반 학생들끼리 모여!"라고 한다. (`GROUP BY`: 그룹화)
4. 반별로 모인 그룹들을 보며 선생님이 마지막으로 "이 중에서, 반 평균 점수가 80점 이상인 그룹만 남아!"라고 외친다. (`HAVING`: 그룹 필터링)

`WHERE`는 그룹화 이전에 개개인을 걸러내는 조건이고, `HAVING`은 그룹화 이후에 그룹 자체를 걸러내는 조건이다.

종합 예제: `WHERE`와 `HAVING` 함께 사용하기

이제 `WHERE`와 `HAVING`을 함께 사용하는 실전 예제를 풀어보자.

마케팅팀에서 새로운 요청이 들어왔다. "**가격이 10만 원 이상인 고가 상품들 중에서만, 카테고리별로 묶었을 때, 그 고가 상품이 2건 이상 팔린** 카테고리는 어디인가요?"

이 요청은 3가지로 분리해서 분석할 수 있다.

- `WHERE`: 가격이 10만 원 이상인 고가 상품들
- `GROUP BY`: 카테고리별로 묶었을 때
- `HAVING`: 그 고가 상품이 2건 이상 팔린 카테고리

이 질문에는 두 가지 필터링 조건이 있다.

1. 개별 주문 건에 대한 필터링: `가격 >= 100000`
2. 그룹화된 결과에 대한 필터링: `주문 건수 >= 2`

이것이 바로 `WHERE`와 `HAVING`을 함께 사용해야 하는 시나리오다.

```
SELECT
    category,
    COUNT(*) AS premium_order_count
FROM
    order_stat
WHERE
    price >= 100000     -- 1. 먼저 개별 행을 가격으로 필터링한다.
GROUP BY
    category            -- 2. 필터링된 행들을 카테고리별로 그룹화한다.
HAVING
    COUNT(*) >= 2;       -- 3. 그룹화된 결과 중, 건수가 2개 이상인 그룹만 필터링한다.
```

쿼리 실행 과정 추적

이 쿼리가 단계별로 어떻게 데이터를 걸러내고 집계하는지 눈으로 직접 따라가 보자.

1단계: `WHERE price >= 100000`

`WHERE` 절이 개별 행들을 필터링한다. `price`가 100,000원 미만인 행들은 이 단계에서 모두 탈락한다.

```
SELECT * FROM order_stat
WHERE price >= 100000;
```

order_id	customer_name	category	product_name	price	quantity	order
1	이순신	전자기기	프리미엄 기계식 키보드	150000	1	2025
3	신사임당	가구	인체공학 사무용 의자	250000	1	2025
5	세종대왕	전자기기	4K 모니터	450000	1	2025
7	이순신	문구	고급 만년필 세트	200000	1	2025
8	세종대왕	가구	높이조절 스탠딩 데스크	320000	1	2025

9	신사임당	전자기기	노이즈캔슬링 블루투스 이어폰	180000	1	2025

- 전체 11개 중 총 6개의 행만 다음 단계로 넘어간다.
- `order_id`: 2,4,6,10,11 주문이 이 단계에서 제외된다.

2단계: `GROUP BY category`
2단계를 통과한 6개의 행을 `category` 기준으로 그룹으로 묶는다.
전자기기, 가구, 문구 그룹으로 묶인다.

그룹1: 전자기기

order_id	customer_name	category	product_name	price	quantity	order
1	이순신	전자기기	프리미엄 기계식 키보드	150000	1	2025
5	세종대왕	전자기기	4K 모니터	450000	1	2025
9	신사임당	전자기기	노이즈캔슬링 블루투스 이어폰	180000	1	2025

그룹2: 가구

order_id	customer_name	category	product_name	price	quantity	order
3	신사임당	가구	인체공학 사무용 의자	250000	1	2025
8	세종대왕	가구	높이조절 스탠딩 데스크	320000	1	2025

그룹3: 문구

order_id	customer_name	category	product_name	price	quantity	order
7	이순신	문구	고급 만년필 세트	200000	1	2025-

3단계: `HAVING COUNT(*) >= 2`
`GROUP BY`로 만들어진 그룹 결과에 대해 `HAVING` 조건절을 적용하여 필터링한다. 각 그룹의 `COUNT(*)` 값이 2 이상인 그룹만 살아남는다. '문구' 그룹은 `COUNT(*)`가 1이므로 이 조건을 만족하지 못해 여기서 제외된다.

필터링 전

category	COUNT(*)
전자기기	3
가구	2
문구	1

필터링 후

category	COUNT(*)
전자기기	3
가구	2

최종 단계: `SELECT ...`
최종적으로 3단계를 통과한 그룹들에 대해 `SELECT` 절에 명시된 컬럼(`category`, `COUNT(*)`)을 선택하여 최종 결과를 화면에 보여준다.

최종 실행 결과

category	premium_order_count
전자기기	3
가구	2

이처럼 `WHERE`는 그룹화할 대상을 미리 선별하는 역할을, `HAVING`은 그룹화된 결과물을 최종적으로 필터링하는 역할을 수행한다.

SQL 실행 순서

우리는 이제 `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `HAVING` 등 다양한 SQL 절을 배웠다. 이들을 조합하여 꽤 복잡한 쿼리를 작성할 수 있게 되었다. 그런데 다음 쿼리를 한번 보자.

"고객별 총 구매 금액을 구하는데, 총 구매 금액이 40만원 이상인 고객만 보려고 한다."

```
SELECT
    customer_name,
    SUM(price * quantity) AS total_purchase
FROM
    order_stat
WHERE
    total_purchase >= 400000 -- 여기! SELECT에서 만든 별칭을 사용했다.
GROUP BY
    customer_name;
```

[실행 결과]

```
Error Code: 1054. Unknown column 'total_purchase' in 'where clause'
```

이 쿼리는 실행하면 오류가 발생한다. `WHERE` 절에서 `total_purchase`라는 컬럼을 찾을 수 없다는 내용의 오류다. 왜일까? `SELECT` 절에서 `AS`를 이용해 `total_purchase` 라는 별칭까지 만들어줬는데, 바로 아래 `WHERE` 절에서는 왜 알아듣지 못하는 걸까?

이 모든 의문은 **우리가 코드를 작성하는 순서**와 **SQL이 실제로 쿼리를 처리하는 순서(논리적 실행 순서)**가 다르기 때문에 발생한다.

SQL 쿼리의 논리적 실행 순서

우리가 코드를 작성하는 순서는 보통 `SELECT` → `FROM` → `WHERE` ... 이지만, 데이터베이스는 다음의 논리적 순서에 따라 쿼리를 해석하고 실행한다.

1. `FROM`: 가장 먼저 실행된다. 어떤 테이블에서 데이터를 가져올지 결정한다. SQL 여정의 출발점이다.
2. `WHERE`: `FROM`에서 가져온 테이블의 **'개별 행'**을 필터링한다. `GROUP BY`로 묶이기 전, 날 것 그대로의 데이터를 1차로 걸러내는 단계다.
3. `GROUP BY`: `WHERE` 절의 필터링을 통과한 행들을 기준으로 그룹을 형성한다.
4. `HAVING`: `GROUP BY`를 통해 만들어진 **'그룹'**들을 필터링한다. `WHERE`가 개인전이라면 `HAVING`은 단체전이다. 집계 함수를 이용한 조건 필터링이 여기서 이루어진다.
5. `SELECT`: 드디어 `SELECT` 절이 실행된다. `HAVING` 절까지 통과한 최종 그룹들에 대해 우리가 보고자 하는 컬럼을 선택하고, `SUM`, `COUNT` 같은 집계 함수 계산, 별칭(`AS`) 부여 등이 모두 이 단계에서 이루어진다.
6. `ORDER BY`: `SELECT` 절에서 선택된 최종 결과 후보들을 지정된 순서로 정렬한다.
    - `SELECT`가 `ORDER BY`보다 먼저 실행되기 때문에, `ORDER BY` 절에서는 `SELECT` 절에서 만든 별칭을 사용할 수 있다. (`ORDER BY total_purchase DESC` 와 같은 구문이 가능한 이유다.)
7. `LIMIT`: 정렬된 결과 중에서 최종적으로 사용자에게 반환할 행의 개수를 제한한다.

이 순서는 반드시 기억하자.

- `1.FROM` → `2.WHERE` → `3.GROUP BY` → `4.HAVING` → `5.SELECT` → `6.ORDER BY` → `7.LIMIT`

이제 `WHERE` 절에서 `SELECT`의 별칭을 쓸 수 없는 이유가 명확해진다. `WHERE` 절을 사용하는 시점에는 아직 `SELECT` 절이 실행되기 한참 전이라, `total_purchase` 같은 별칭은 세상에 존재하지도 않기 때문이다.

> **SQL의 논리적인 실행 순서와 물리적인 실행 순서**
> SQL의 **논리적 실행 순서**에 맞추어 쿼리를 작성해야 의도한 결과를 정확히 얻을 수 있다.
> 데이터베이스 엔진은 성능 최적화를 위해 내부적으로 이 순서를 재배열해 **물리적 실행 순서**를 결정한다.
> 그러나 물리적 순서가 바뀌더라도 엔진은 동일한 결과를 보장하므로, 사용자는 논리적 실행 순서에 맞추어 쿼리를 작성하면 된다.

실전 예제로 따라가는 실행 순서

이제 우리 쇼핑몰 데이터로 복잡한 쿼리를 하나 작성하고, 앞서 배운 7단계 실행 순서에 따라 데이터가 어떻게 가공되는지 단계별로 추적해 보자.

문제

2025년 5월 14일 이전에 들어온 주문들 중에서(WHERE), 고객별로 그룹화하여(GROUP BY), 주문 건수가 2회 이상인 고객을 찾아서(HAVING), 해당 고객의 이름과 총 구매 금액을 조회하고(SELECT), 총 구매 금액을 기준으로 내림차순 정렬해라(ORDER BY) 그리고 하나의 데이터만 출력해라.

```sql
SELECT
    customer_name,
    SUM(price * quantity) AS total_purchase -- 5단계
FROM
    order_stat -- 1단계
WHERE
    order_date < '2025-05-14' -- 2단계
GROUP BY
    customer_name -- 3단계
HAVING
    COUNT(*) >= 2 -- 4단계
ORDER BY
    total_purchase DESC -- 6단계
LIMIT 1; -- 7단계
```

단계별 추적

데이터가 이 쿼리를 만나 어떻게 여행하는지 따라가 보자.

1단계: `FROM order_stat`

우선 `order_stat` 테이블에 있는 11개의 모든 행을 작업대로 가져온다.

2단계: `WHERE order_date < '2025-05-14'`

가져온 11개 행을 하나씩 검사하여 `order_date`가 '2025-05-14' 이전인지 확인한다.

- 5월 14일 이후의 주문 5건(`이순신-만년필`, `세종대왕-데스크`, `신사임당-이어폰`, `장영실-보조배터리`, `홍길동-USB허브`)이 **제외**된다. 6개의 행이 다음 단계로 넘어간다.

```sql
SELECT * FROM order_stat
WHERE order_date < '2025-05-14';
```

남은 데이터 (6개 행)

customer_name	product_name	order_date	...

이순신	프리미엄 기계식 키보드	2025-05-10	...
세종대왕	SQL 마스터링	2025-05-10	...
신사임당	인체공학 사무용 의자	2025-05-11	...
이순신	고성능 게이밍 마우스	2025-05-12	...
세종대왕	4K 모니터	2025-05-12	...
장영실	파이썬 데이터 분석	2025-05-13	...

3단계: `GROUP BY customer_name`
남아있는 6개 행을 `customer_name` 기준으로 묶어 그룹을 만든다.

- **Group 1 (이순신)**: 키보드 주문, 마우스 주문 (2개 행)
- **Group 2 (세종대왕)**: SQL책 주문, 모니터 주문 (2개 행)
- **Group 3 (신사임당)**: 의자 주문 (1개 행)
- **Group 4 (장영실)**: 파이썬책 주문 (1개 행)

총 4개의 그룹이 생성되었다.

4단계: `HAVING COUNT(*) >= 2`
생성된 4개의 그룹을 대상으로, 각 그룹의 행 개수(`COUNT(*)`)가 2 이상인지 검사한다.

- **Group 1 (이순신)**: `COUNT(*)` 는 2. 조건 일치, **통과**.
- **Group 2 (세종대왕)**: `COUNT(*)` 는 2. 조건 일치, **통과**.
- **Group 3 (신사임당)**: `COUNT(*)` 는 1. 조건 불일치, **제외**.
- **Group 4 (장영실)**: `COUNT(*)` 는 1. 조건 불일치, **제외**.

오직 '이순신'과 '세종대왕' 그룹 두 개만이 살아남았다.

5단계: `SELECT customer_name, SUM(...) AS total_purchase`
최종적으로 살아남은 두 그룹에 대해 `SELECT` 절을 실행한다.

- **'이순신' 그룹**: `SUM` 계산: 150000 * 1 + 80000 * 1 = 230000
- **'세종대왕' 그룹**: `SUM` 계산: 35000 * 2 + 450000 * 1 = 520000

이제 최종 결과 집합의 후보로 다음 두 행이 만들어졌다.

customer_name	total_purchase
이순신	230000
세종대왕	520000

6단계: `ORDER BY total_purchase DESC`
`SELECT` 된 두 행을 `total_purchase` 기준으로 내림차순 정렬한다. 520,000이 230,000보다 크므로 '세종대왕' 행이 먼저 온다.

customer_name	total_purchase
세종대왕	520000
이순신	230000

7단계: `LIMIT 1`**, 최종 결과**
하나의 행만 선택한다.

customer_name	total_purchase
세종대왕	520000

SQL의 논리적 실행 순서를 따라가본 덕분에, 복잡해 보였던 쿼리가 한 단계 한 단계 어떻게 작동하는지 이해 되었을 것이다.

문제와 풀이

문제1: 기본 통계 조회하기

[문제]
`order_stat` 테이블을 사용하여 쇼핑몰의 전체 주문 건수와, 카테고리 정보가 누락되지 않은(NULL이 아닌) 주문 건수를 각각 조회해라. 컬럼의 별칭은 각각 '총 주문 건수', '카테고리 보유 건수'로 지정해라. 실행 결과를 참고해라.

[실행 결과]

총 주문 건수	카테고리 보유 건수
11	10

[정답]

```
SELECT
    COUNT(*) AS `총 주문 건수`,
    COUNT(category) AS `카테고리 보유 건수`
FROM
    order_stat;
```

문제2: 쇼핑몰 매출 현황 파악하기

[문제]

`order_stat` 테이블을 사용하여 우리 쇼핑몰의 총 매출액, 평균 주문 금액(주문 1건당 매출액), 판매된 상품들의 최고 단가와 최저 단가를 한 번에 조회해라. 실행 결과를 참고해라.

- 총 매출액: `price` * `quantity` 의 합계
- 평균 주문 금액: `price` * `quantity` 의 평균
- 최고 단가: `price`의 최댓값
- 최저 단가: `price`의 최솟값

[실행 결과]

총 매출액	평균 주문 금액	최고 단가	최저 단가
1985000	180454.5455	450000	35000

[정답]

```
SELECT
    SUM(price * quantity) AS `총 매출액`,
    AVG(price * quantity) AS `평균 주문 금액`,
    MAX(price) AS `최고 단가`,
    MIN(price) AS `최저 단가`
```

```
FROM
    order_stat;
```

문제3: 카테고리별 실적 분석하기

[문제]

`order_stat` 테이블을 `category` 별로 그룹화하여, 각 카테고리별로 총 판매된 상품 수량(`quantity`의 합계)과 총 매출액(`price * quantity`의 합계)을 계산해라. 결과는 총 매출액이 높은 순서대로 정렬해야 한다. 실행 결과를 참고해라.

[실행 결과]

category	카테고리별 총 판매 수량	카테고리별 총 매출액
전자기기	6	960000
가구	2	570000
문구	1	200000
도서	5	190000
NULL	1	65000

[정답]

```
SELECT
    category,
    SUM(quantity) AS `카테고리별 총 판매 수량`,
    SUM(price * quantity) AS `카테고리별 총 매출액`
FROM
    order_stat
GROUP BY
    category
ORDER BY
    `카테고리별 총 매출액` DESC;
```

> ORDER BY에서 SELECT 절의 별칭 컬럼을 선택할 때는 백틱(`)을 사용해야 한다.

문제4: 고객별 주문 통계 분석하기

[문제]

`order_stat` 테이블을 사용하여 고객별로 총 주문 횟수와 총 구매한 상품의 수량(`quantity`)을 계산해라. 결과는 주문 횟수가 많은 순으로, 주문 횟수가 같다면 총 구매 수량이 많은 순으로 정렬해라. 실행 결과를 참고해라.

[실행 결과]

customer_name	총 주문 횟수	총 구매 수량
세종대왕	3	4
이순신	3	3
장영실	2	5
신사임당	2	2
홍길동	1	1

[정답]

```
SELECT
    customer_name,
    COUNT(*) AS `총 주문 횟수`,
    SUM(quantity) AS `총 구매 수량`
FROM
    order_stat
GROUP BY
    customer_name
ORDER BY
    `총 주문 횟수` DESC, `총 구매 수량` DESC;
```

문제5: VIP 고객 필터링하기

[문제]

`order_stat` 테이블에서 고객별 총 구매 금액을 계산하고, 총 구매 금액이 40만 원 이상인 'VIP 고객' 목록만 조회해라. 결과에는 고객 이름과 총 구매 금액이 포함되어야 하며, 총 구매 금액이 높은 순으로 정렬되어야 한다. 실행 결과를 참고해라.

[실행 결과]

customer_name	총 구매 금액
세종대왕	840000
이순신	430000
신사임당	430000

[정답]

```
SELECT
    customer_name,
    SUM(price * quantity) AS `총 구매 금액`
FROM
    order_stat
GROUP BY
    customer_name
HAVING
    SUM(price * quantity) >= 400000
ORDER BY
    `총 구매 금액` DESC;
```

문제6: 복합 조건으로 핵심 고객 그룹 찾기

[문제]

`order_stat` 테이블에서 '도서' 카테고리를 제외한(`WHERE`) 주문들 중에서, 2회 이상 주문한 고객들을 찾아(`GROUP BY`, `HAVING`), 그 고객들의 이름, 주문 횟수, 총 사용 금액을 조회해라. 결과는 총 사용 금액이 높은 순으로 정렬되어야 한다.

이 문제는 SQL 실행 순서와 `WHERE`, `HAVING`의 차이점을 잘 이해해야 풀 수 있다. 실행 결과를 참고해라.

[실행 결과]

customer_name	주문 횟수	총 사용 금액
세종대왕	2	770000
이순신	3	430000
신사임당	2	430000

[정답]

```
SELECT
    customer_name,
    COUNT(*) AS `주문 횟수`,
    SUM(price * quantity) AS `총 사용 금액`
FROM
    order_stat
WHERE
    category != '도서' OR category IS NULL -- '도서' 카테고리가 아닌 주문, NULL도 포함
GROUP BY
    customer_name
HAVING
    COUNT(*) >= 2
ORDER BY
    `총 사용 금액` DESC;
```

정리

집계 함수

- `COUNT(*)`는 `NULL` 값을 포함한 테이블의 전체 행 개수를 반환한다.
- `COUNT(컬럼명)`은 해당 컬럼에서 `NULL`이 아닌 값의 개수만 반환한다.
- `SUM()`과 `AVG()`는 숫자 컬럼의 합계와 평균을 계산하며 `NULL` 값은 자동으로 제외한다.
- `MAX()`와 `MIN()`은 컬럼에서 최댓값과 최솟값을 찾는다.
- `DISTINCT`는 `COUNT`와 함께 사용되어 중복을 제외한 고유한 값의 개수를 센다.

GROUP BY - 그룹으로 묶기

- `GROUP BY`는 특정 컬럼의 값이 같은 행들을 하나의 그룹으로 묶어주는 역할을 한다.
- 각 그룹에 집계 함수를 적용하여 그룹별 통계를 계산할 수 있다.
- 여러 컬럼을 기준으로 그룹화하여 더 세분화된 분석이 가능하다.
- `GROUP BY`는 `NULL` 값도 하나의 독립된 그룹으로 취급하여 집계한다.

GROUP BY - 주의사항

- `GROUP BY`를 사용하면 `SELECT` 절에는 `GROUP BY`에 사용된 컬럼이나 집계 함수만 올 수 있다.
- 그룹으로 묶으면 여러 행이 하나의 행으로 요약되므로 그룹을 대표할 수 없는 개별 행의 컬럼은 조회할 수 없다.

HAVING - 그룹 필터링1

- `HAVING` 절은 `GROUP BY`로 그룹화된 결과에 대해 조건을 적용하여 필터링하는 역할을 한다.
- `WHERE` 절은 그룹화 전에 개별 행을 필터링하므로 집계 함수를 사용할 수 없다.
- `HAVING` 절의 조건문에는 `SUM()` `COUNT()` 같은 집계 함수를 사용할 수 있다.
- SQL 실행 순서상 `GROUP BY` 다음에 `HAVING`이 실행된다.

HAVING - 그룹 필터링2

- `WHERE`와 `HAVING`은 필터링 대상과 시점이 다르다.
- `WHERE`는 그룹화 전 개별 행을 대상으로 하고 `HAVING`은 그룹화 후 생성된 그룹을 대상으로 한다.
- 개별 행에 대한 조건은 `WHERE`에 그룹에 대한 조건은 `HAVING`에 작성하여 함께 사용할 수 있다.

SQL 실행 순서

- SQL 쿼리는 코드 작성 순서가 아닌 정해진 논리적 순서에 따라 실행된다.
- 논리적 실행 순서는 `FROM` → `WHERE` → `GROUP BY` → `HAVING` → `SELECT` → `ORDER BY` → `LIMIT` 순서이다.
- 이 실행 순서 때문에 `WHERE` 절에서는 `SELECT` 절에서 만든 별칭을 사용할 수 없지만 `ORDER BY` 절에서는 사용할 수 있다.