[베이직반] SQL 2회차

구분	선택 학습반
날짜	@2025년 9월 29일 오후 7:00
대상	베이직
태그	SQL
튜터	강 강민정

[강의 자료]

베이직반_SQL_2회차_수업자료.pdf

[강의 녹화본]

📖 목차

1. JOIN 개념 및 사용 방법

👋 참고 자료: https://persistent-polyanthus-3e0.notion.site/SQL-4-24722d577d0e806fa196c3c68a454ba9?pvs=74

1-1. JOIN을 하는 방법

Table - s1

name(PK)	age
alice	10
angella	15
barbara	13
Emma	11
Issabel	10
Alexander	9
David	16
Gabriel	17
Henry	19

Table - s2

name(FK)	goods_nm	goods_pay_date
alice	subscribe_package	2024-01-06
angella	premium_package	2024-02-17
barbara	premium_package	2023-12-01
Emma	subscribe_package	2023-11-07
Issabel	new_package	2024-03-25
Winifred	new_package	2023-12-12

1️⃣ JOIN의 첫 번째 단계: 공통컬럼 찾기

- 조인을 위해서는 공통컬럼을 먼저 찾아야 합니다.
- 공통컬럼은 곧 테이블과 테이블의 연결 고리로 작용하게 됩니다.
  - 공통컬럼 = 두 테이블에서 공통으로 존재하는 컬럼(열)
  - 위 예제에서는 `name` 이 공통컬럼이네요!
  - 예제에서는 공통컬럼의 이름이 같았지만, 이름이 달라도 조인이 가능합니다.
    - 단, 조인 조건에서 명시적으로 지정해야 합니다. 예시) `ON a.id = b.user_id`
- **공통컬럼이 없다면, 조인을 할 수 없다**는 점을 기억해주세요!

1. 공통컬럼이 1개인 경우 기본 쿼리 작성법

🤓 **SELECT → FROM → JOIN/ON (필요하다면) → WHERE → GROUP BY → HAVING → ORDER BY → LIMIT**

```
-- 모든 컬럼을 다 가져올 때
-- 테이블 구조와 데이터를 한 번에 다 확인 가능하지만 불필요한 컬럼이 포함될 수 있음.

SELECT *
FROM 테이블 AS a
JOIN 테이블 AS b
  ON a.공통컬럼 = b.공통컬럼;


-- 필요한 컬럼을 지정해서 추출할 때
-- 성능 좋음, 가독성 높음, 불필요한 데이터 줄임

SELECT
    a.컬럼1,
    b.컬럼2
FROM 테이블 AS a
JOIN 테이블 AS b  -- JOIN 종류: INNER, LEFT, RIGHT 등 지정
    ON a.공통컬럼 = b.공통컬럼;
```

2. 공통컬럼이 2개 이상인 경우 기본 쿼리 작성법

🤓 **SELECT → FROM → JOIN/ON (필요하다면) → WHERE → GROUP BY → HAVING → ORDER BY → LIMIT**

```
-- 모든 컬럼을 다 가져올 때
-- 테이블 구조와 데이터를 한 번에 다 확인 가능하지만 불필요한 컬럼이 포함될 수 있음.

SELECT *
FROM 테이블 AS a
JOIN 테이블 AS b  -- JOIN 종류: INNER, LEFT, RIGHT 등 지정
    ON a.공통컬럼1 = b.공통컬럼1
   AND a.공통컬럼2 = b.공통컬럼2;



-- 필요한 컬럼을 지정해서 추출할 때
-- 성능 좋음, 가독성 높음, 불필요한 데이터 줄임

SELECT
    a.컬럼1,
    b.컬럼2
FROM 테이블 AS a
JOIN 테이블 AS b  -- JOIN 종류: INNER, LEFT, RIGHT 등 지정
    ON a.공통컬럼1 = b.공통컬럼1
   AND a.공통컬럼2 = b.공통컬럼2;
```

2️⃣ JOIN의 두 번째 단계: 공통컬럼 관계찾기(PK와 FK 찾기)

- 공통컬럼을 찾았다면 두 번째로 어떤 공통컬럼이 기준이 되고, 어떤 공통컬럼이 종속되어 있는지 파악해야 합니다.
- 우리는 이를 `PK`, `FK` 라는 단어로 약속했습니다.

🤓
- 테이블A: 나는 이름컬럼과 나이컬럼이 있어 + 내 이름컬럼이 기준이야!(PK)
- 테이블B: 나는 이름컬럼과 나이컬럼과 국가컬럼이 있어 + 내 이름컬럼은 테이블A 이름컬럼을 참조할꺼야 (FK)

▼ 이러한 관계들은 **데이터 분석가가 설정하는 것이 아닌, 데이터 수집/저장 단계에서 정해집니다.**

- PK(기본키)와 FK(외래키)는 **데이터 설계 단계**에서 데이터베이스 관리자(DBA)나 **개발자**가 미리 정의합니다.
- 이건 데이터가 **수집·저장될 때부터** 구조(스키마)에 포함돼 있어요.
- 분석가가 임의로 PK/FK를 바꾸면, 다른 시스템이나 애플리케이션 동작이 깨질 수 있기 때문에 **분석가가 설정하지 않습니다**.

✅ **분석가의 역할**

- 이미 정해진 PK/FK 관계를 **이해하고**, 그걸 **JOIN 조건에 활용**합니다.
- PK/FK 관계를 알면
  - 어떤 테이블이 기준(주인, PK)인지
  - 어떤 테이블이 종속(손님, FK)인지
  - 어떤 방식으로 테이블을 합쳐야 데이터가 중복되거나 누락되지 않는지를 쉽게 파악할 수 있습니다.

- **데이터 분석가는 공통컬럼의 기준과 비교값을 찾아 테이블 관계를 확인할 수 있습니다.**
- 따라서 이렇게 명시되어 있는 관계를 잘 파악하는 것이 중요합니다. 개념을 짚어보겠습니다.

구분	상세 설명
PK (Primary Key, 기본키)	• `NULL` **일 수 없고, 유일한 값을 가짐.** • 모든 데이터를 식별하는 기준이 되는 컬럼 • 테이블 당 하나의 기본키만 가질 수 있음.
FK (Foreign Key, 외래키)	• 다른 테이블의 PK와 연결되어 테이블 간 관계를 나타내는 컬럼 • 기준이 되는 컬럼(PK)을 확인하기 위한 연결 컬럼의 역할

- PK(기본키)와 FK(외래키)는 **다른 테이블을 연결하기 위해 사용되는 "고리" 같은 역할**
- 우리는 이것을 ERD(Entity Relationship Diagram)라는 관계도를 통해서 확인할 수 있습니다.
  - ERD 내 기호들을 통해서, 각 테이블이 어떻게 대응되는 지 파악할 수 있는데요. 이를 `Mapping Cardinality` 라고 부릅니다.

- 예시)

Table - s1

name(PK)	age
alice	10
angella	15
barbara	13
Emma	11
Issabel	10
Alexander	9
David	16
Gabriel	17
Henry	19

Table - s2

name(FK)	goods_nm	goods_pay_date
alice	subscribe_package	2024-01-06
angella	premium_package	2024-02-17
barbara	premium_package	2023-12-01
Emma	subscribe_package	2023-11-07
Issabel	new_package	2024-03-25
Winifred	new_package	2023-12-12

- s1 테이블에는 name과 age가 있고, name 이 PK로 되어있네요!
- s2 테이블은 name, good_nm, goods_pay_date가 있고, name이 FK 로 되어있어요!
- 그렇다면 기준 테이블은 s1이 됩니다.

▼ 왜죠?

1. **s1 테이블**
   - `name`이 기본키(PK)예요.
   - 즉, 이 테이블에서는 `name`이 **고유한 사람을 구분해주는 핵심 컬럼**이라는 뜻이에요.
2. **s2 테이블**
   - `name`이 외래키(FK)예요.
   - 즉, 이 테이블은 `name`을 통해 **s1 테이블의 사람 정보를 참조**하고 있다는 뜻이에요.
   - 예시) "이 결제는 s1에 있는 누구누구의 것이다!"라고 알려주는 역할
3. 따라서 **두 테이블을 조인할 때**
   - `s1`이 사람의 "기본 정보" 테이블이고,
   - `s2`는 그 사람에 대한 "결제 이력"을 가진 테이블이므로,
   - **s1을 기준 테이블로 하고, s2와 조인하면 되겠구나!**

- `s1` 을 예를 들어서 **학교에서 관리하는 학생 명단**이라고 칩시다. 이름은 고유하고 중복되지 않죠.
- `s2`는 **학생들이 어떤 학습 상품을 결제했는지**를 기록한 표라고 가정합시다. 한 학생이 여러 번 결제할 수 있기 때문에 **이름이 여러 번 나올 수 있어요.**

🍀 그래서, 어떤 학생이 뭘 결제했는지를 보려면, **먼저 학생 명단(s1)을 기준으로 삼고**, `s2`의 결제정보를 **이름(name)을 기준으로 붙여야** 해요.

→ s1은 신뢰할 수 있는 공식 명단(기준)이고, s2는 결제가 있는 사람만 있으니까, 누가 결제했는지 정확히 보려면 기준이 되는 s1에 s2를 조인해야 한다는 뜻이에요.

- 그럼 데이터 분석가로서 어떻게 JOIN 기준을 정해야 할까?
  - 기준 테이블이 **s1**이라는 걸 알았을 때, **어떤 JOIN 방식을 써야 하는지** 결정하는 건 **"우리가 보고 싶은 결과가 무엇인가?"**, 즉 **분석 목적**에 따라 결정하면 됩니다.

3️⃣ 조인의 세번째 단계: 적절한 조인 방식 찾기

조인 종류	설명	반환 데이터 범위	MySQL 지원 여부
INNER JOIN(가장 많이 사용)	두 테이블에서 **일치하는 값**을 가진 행만 반환 (교집합)	두 테이블의 교집합	지원
LEFT JOIN(가장 많이 사용)	왼쪽 테이블의 **모든 행** + 오른쪽 테이블에서 일치하는 행 반환. 일치하지 않으면 오른쪽 컬럼은 NULL	왼쪽 전체 + 교집합	지원
RIGHT JOIN	오른쪽 테이블의 **모든 행** + 왼쪽 테이블에서 일치하는 행 반환. 일치하지 않으면 왼쪽 컬럼은 NULL	오른쪽 전체 + 교집합	지원
FULL OUTER JOIN	양쪽 테이블의 **모든 행** 반환 (합집합). 일치하지 않는 컬럼은 NULL로 채움	합집합 (교집합 + 왼쪽/오른쪽 단독 데이터)	MySQL 기본적으로 미지원 → `LEFT JOIN` 과 `RIGHT JOIN` 의 합집합으로 계산해야 함.

1-2. JOIN 방식

✅ INNER JOIN

- INNER JOIN 기본 구문 패턴 익히기

```
SELECT
   a.컬럼1,
   b.컬럼2 ...
FROM 테이블명1 AS a
INNER JOIN 테이블명2 AS b
   ON a.공통컬럼 = b.공통컬럼;
```

✅ LEFT JOIN

- LEFT JOIN 기본 구문 패턴 익히기

```
SELECT                  -- 모든 컬럼을 다 가져올 때는 * 처리 / 컬럼명에는 별칭.컬럼명 이런식으로 해주셔야해요.
   a.컬럼1,
   b.컬럼2,
   ...
FROM
      테이블1 AS a            -- 기준 테이블 (왼쪽)
LEFT JOIN
      테이블2 AS b            -- 조인할 테이블 (오른쪽)
   ON a.공통컬럼 = b.공통컬럼     -- 조인 조건
```

```
SELECT                  -- 모든 컬럼을 다 가져올 때는 * 처리 / 컬럼명에는 별칭.컬럼명 이런식으로 해주셔야해요.
   a.컬럼1,
   b.컬럼2,
   ...
FROM 테이블1 AS a            -- 기준 테이블 (왼쪽)
LEFT JOIN 테이블2 AS b        -- 조인할 테이블 (오른쪽)
    ON a.공통컬럼 = b.공통컬럼    -- 조인 조건
```

▼ 쿼리 해석

- `FROM 테이블1 AS a`: 기준이 되는 테이블을 `a`라는 별칭으로 사용합니다.
- `LEFT JOIN 테이블2 AS b`: `테이블1`의 모든 행을 기준으로 `테이블2`를 조인합니다. 매칭되는 값이 없으면 `NULL`이 들어갑니다.
- `ON a.공통컬럼 = b.공통컬럼`: 두 테이블 간 조인 조건을 명시합니다.

어떤 조인이든지 간에,

`JOIN 텍스트` 위에 있으면 LEFT에 위치한 테이블, `JOIN 텍스트` 아래(옆)에 있으면 RIGHT!!!

근데 LEFT JOIN이라면서요, 그럼 LEFT가 기준이니까 `테이블1`이 기준 테이블이죠 😎

왼쪽에 테이블은 가만히 있고, 오른쪽에 있는 테이블을 갖다 붙인다!

- **LEFT JOIN 에서 기준이 되는 테이블**(`테이블1`)
  = **LEFT 텍스트를 기준으로 위쪽에 작성하는 테이블** (`테이블1`)
  = **FROM 아래(옆)에 작성되는 테이블** (`테이블1`)
  = **LEFT에 위치해 있는 테이블** (`테이블1`)
- 테이블1, 테이블2 중에 어떤 것을 LEFT 테이블(기준 테이블) 로 잡는 게 좋을까요?
  - **보통 전체 데이터(모든 행을 포함해야 하는 테이블)를 LEFT 테이블로 잡습니다.**
- LEFT에 위치한 (기준이 되는) `테이블1` 은 조인 조건을 만족해도 or 만족하지 못해도 모두 출력됩니다.

- **LEFT JOIN 에서 조건에 따라 출력되는 테이블**(`테이블2`)
  = **LEFT 텍스트 기준으로 아래에 작성하는 테이블**(`테이블2`)
  = **LEFT JOIN 텍스트 아래(옆)에 작성되는 테이블** (`테이블2`)
  = **RIGHT에 위치해 있는 테이블** (`테이블2`)
- RIGHT에 위치한 `테이블2`은 조인 조건을 만족하는 경우 출력되며, 만족하지 못할 경우 NULL 값으로 출력됩니다.

2. WITH 구문

하나의 테이블이 여러 번 필요해요!

참고 자료: https://persistent-polyanthus-3e0.notion.site/SQL-5-24722d577d0e806694a8f10af9bfae5b

구분	상세
정의	SQL 구문에서 사용되는 **임시 테이블(가상 테이블)**
사용 이유	쿼리의 **가독성 향상** 및 **쿼리 성능 최적화**
특징 & 장점	- 임시 테이블처럼 사용되며, **작성한 쿼리 내에서만 유효** - **여러 개의 WITH 문 선언 가능** - **한 테이블을 여러 번 조회**해야 하는 경우, 1회만 조회해 성능 향상 (한 번만 저장해놓으면 계속 쓸 수 있음) - 복잡한 **JOIN, UNION 등 연산을 효율적으로 처리**

- WITH 구문 기초 작성 방법

```
WITH 임시테이블명 AS (
    SELECT 컬럼1, 컬럼2, ...
    FROM 원본테이블명
    WHERE 조건
)
SELECT 원하는컬럼1, 원하는컬럼2, ...
FROM 임시테이블명;
```

```
WITH 임시테이블명 AS (
    SELECT
        컬럼1,
        컬럼2,
        집계함수(컬럼3) AS 집계값,
        윈도우함수() OVER (
            PARTITION BY 컬럼1
            ORDER BY 컬럼2
        ) AS 윈도우값
    FROM 원본테이블명
    WHERE 조건식
    GROUP BY 컬럼1, 컬럼2
    HAVING 집계조건식
)
SELECT
    원하는컬럼1,
    원하는컬럼2,
    ...
FROM 임시테이블명
ORDER BY 정렬기준컬럼 ASC;
```

✅ WITH 구문 예시

- 예시 1 - 기본적인 WITH 절 사용

```
-- with 절로 임시 테이블 soso 정의
WITH soso AS (
    SELECT
        etc_str2,
        etc_str1,
        COUNT(DISTINCT game_actor_id) AS actor_cnt
    FROM
        basic.users
```

```
    GROUP BY
        etc_str2,
        etc_str1
)

-- 임시 테이블 soso 사용
SELECT *
FROM soso;
```

- 예시 2 - 경험치가 가장 많은 캐릭터 정보 조회

```
-- 임시 테이블 dodo 정의
WITH dodo AS (
    SELECT *
    FROM basic.users
)

-- max 경험치 보유한 캐릭터 정보 추출
SELECT *
FROM (
    SELECT MAX(exp) AS maxexp
    FROM dodo
) AS a
INNER JOIN (
    SELECT *
    FROM dodo
) AS b
ON a.maxexp = b.exp;
```

- 예시 3 - 다중 WITH 구문 사용 예시

```
-- 첫 번째 임시 테이블 gogo 정의: 레벨 50 초과 유저
WITH gogo AS (
    SELECT
        game_account_id,
        exp
    FROM
        basic.users
    WHERE
        `level` > 50
),

-- 두 번째 임시 테이블 hoho 정의: 카드 결제 정보
hoho AS (
    SELECT DISTINCT
        game_account_id,
        pay_amount,
        approved_at
    FROM
        basic.payment
    WHERE
        pay_type = 'CARD'
)

-- 결제 여부별 유저 수 집계
```

```
SELECT
   CASE
      WHEN b.game_account_id IS NULL THEN '결제x'
      ELSE '결제o'
   END AS gb,
   COUNT(DISTINCT a.game_account_id) AS accnt
FROM
   gogo AS a
LEFT JOIN
   hoho AS b
ON
   a.game_account_id = b.game_account_id
GROUP BY
   CASE
      WHEN b.game_account_id IS NULL THEN '결제x'
      ELSE '결제o'
   END;
```

▼ **다중 WITH 구문 사용 시 주의할 점** 🚨

- SQL에서 **WITH (CTE)** 구문은 보통 **맨 위에 한 번만** 선언합니다.
- 여러 개의 CTE가 필요하다면 아래처럼 **콤마(,)로 이어서** 한 번의 WITH 블록 안에 정의하는 게 일반적이고 표준이죠.

```
WITH A AS (
   SELECT ...
),
B AS (
   SELECT ...
),
C AS (
   SELECT ...
)
SELECT *
FROM C;
```

✅ 서브쿼리 vs WITH 문 (공통 테이블 표현식)

항목	서브쿼리	WITH 문 (CTE)
위치	SELECT, FROM, WHERE 절 안에 바로 씀	쿼리의 맨 위에 작성
재사용	안됨	여러 번 참조 가능
가독성	복잡해지기 쉬움	**가독성이 좋고 유지보수에 유리**
사용 예	간단한 조건에 자주 사용	복잡한 로직 처리 시 선호

상황	추천 도구	이유
쿼리가 길고, 중간 계산을 나눠서 작성하고 싶을 때	`WITH 문`	각 단계를 이름 붙여 관리 가능, **가독성↑**
같은 쿼리를 여러 번 써야 할 때	`WITH 문`	중복 제거, **재사용성↑**
쿼리가 짧고 단순한 조건만 필요할 때	서브쿼리	간결하게 처리, 오히려 `WITH` 보다 **짧고 명확**
성능 최적화가 중요할 때	상황에 따라 다름	어떤 RDBMS에서는 서브쿼리가 더 빠를 수도 있음 (즉, **DB 엔진마다 다름**)

▼ **WITH 구문(CTE)을 써야 하는 대표적인 상황**

✅ 1. **같은 서브쿼리를 여러 번 재사용할 때**

- 문제에 이런 표현이 나오면 의심해봅시다.
  - 같은 집계 결과를 여러 번 사용하라.
  - 먼저 ~의 합계를 구한 후, 이를 이용해서 ... 계산하라.
- 예시

```
WITH SalesSummary AS (
   SELECT customer_id, SUM(amount) AS total
   FROM sales
   GROUP BY customer_id
)
SELECT *
FROM SalesSummary
WHERE total > 1000;
```

✅ **2. 문제를 단계별로 쪼개야 할 때**

- 문제에 이런 힌트가 있으면 CTE 후보
  - 먼저 A를 구한 뒤, 그 결과를 이용해서 B를 구하라.
  - 중간 결과를 바탕으로 최종 결과를 출력하라.
- 즉, **중간 단계 → 최종 답 구조**가 나오면 WITH가 잘 맞을 수 있죠.

✅ 3. **가독성을 높이고 싶을 때**

- 문제에 복잡한 조건이나 중첩 서브쿼리가 나오면, CTE로 나누면 보기 편해집니다.
- 예시) 평균 이상을 구하고, 그 결과를 다시 집계 같은 문제 → 이런 경우에는 서브쿼리도 좋지만 CTE로 이름 붙여도 괜찮은 풀이가 될 수 있음.

- 테이블: `employees(emp_id, name, dept_id, salary)`
- 요구사항
  1. 부서별 평균 급여를 구한다.
  2. 각 부서에서 평균 이상 급여를 받는 사원만 뽑는다.
  3. 부서별로 그런 사원의 수를 집계한다.

1. **서브쿼리 버전**

```
SELECT e.dept_id,
       COUNT(*) AS cnt_above_avg
FROM employees e
WHERE e.salary >= (
    SELECT AVG(salary)
    FROM employees
    WHERE dept_id = e.dept_id
)
GROUP BY e.dept_id;
```

▼ 쿼리 해석

- 서브쿼리에서 `dept_id`를 맞춰서 부서별 평균을 구함.
- WHERE에서 다시 비교
- 서브쿼리가 `employees`를 계속 반복해서 읽는 중...

2. **CTE 버전**

```
WITH dept_avg AS (
    SELECT dept_id, AVG(salary) AS avg_salary
    FROM employees
    GROUP BY dept_id
),
above_avg AS (
    SELECT e.dept_id, e.emp_id, e.salary
    FROM employees e
    JOIN dept_avg d ON e.dept_id = d.dept_id
    WHERE e.salary >= d.avg_salary
)
SELECT dept_id,
       COUNT(*) AS cnt_above_avg
FROM above_avg
GROUP BY dept_id;
```

▼ 쿼리 해석

- `dept_avg`: 부서별 평균을 먼저 구해둠
- `above_avg`: 평균 이상인 사원만 걸러서 중간 테이블로 만듦.
- 최종 SELECT에서 그 중간 결과를 다시 집계
- 단계가 분리돼서 **논리 흐름이 눈에 잘 들어오고**, 나중에 중간 CTE(`above_avg`)를 따로 조회해서 검증이 용이함.

✅ 예시

- **서브쿼리**

```
SELECT name
FROM students
WHERE score > (
    SELECT AVG(score) FROM students
);
```

- **WITH 문**

```
WITH avg_score AS (
    SELECT AVG(score) AS avg FROM students
```

```
)
SELECT name
FROM students, avg_score
WHERE score > avg;
```

✅ 실무에서는?

- **복잡한 로직이면 WITH 문 선호** (가독성, 유지보수 측면)
- 단순한 조건만 있는 경우엔 **서브쿼리**도 많이 씁니다.

WITH 문과 서브쿼리는 "상황에 맞는 선택"이 필요하다!

🚨 실무에서는 복잡한 로직이나 재사용이 필요한 경우 WITH 문을 자주 사용하는 편이지만,
간단한 조건 필터링이나 단일 조회에는 서브쿼리로도 충분히 처리합니다.
두 방법 중 **절대적으로 더 좋은 것은 없으며, 상황에 맞게 선택하는 것이 중요**합니다.

3. SQL 문제 풀이의 핵심 키워드 유형

❤️ 문제에서 아래의 키워드들이 나오면 **"아 그럴 수도 있겠구나!!!"** 의심을 해봅시다. 그리고 문제에 적용해봅시다.

3-1. 키워드 정리

1. SELECT 절 관련 키워드

키워드	의미	예시
가장 큰, 최고, 최대	MAX() 사용	가장 큰 점수를 가진 학생
가장 작은, 최소	MIN() 사용	최소 급여를 받는 직원
평균, 평균값	AVG() 사용	평균 나이 이상인 회원
합계, 총	SUM() 사용	총 매출이 100만 이상인 지점

2. WHERE 절 관련 키워드

키워드	의미	예시
~한 사람, 조건을 만족하는, ~이 아닌	WHERE 조건문 필요	나이가 30 이상인 고객
~을 포함, 포함된, ~중 하나	IN, LIKE	'서울', '부산' 거주자
~가 아닌, 제외하고	NOT, !=, NOT IN	관리자 제외

3. JOIN 관련 키워드

키워드	의미	예시
~에 속한, ~의, ~정보와 함께	두 개 이상의 테이블 연결 필요	주문 정보 + 고객 정보
같은, 일치하는	ON 조건 설정 필요	같은 부서의 직원

4. GROUP BY 관련 키워드

키워드	의미	예시
~별로 , 각 , ~당	GROUP BY	부서별 평균 급여
가장 많이 , 가장 적게	GROUP BY + COUNT() + ORDER BY DESC	가장 주문이 많은 제품

5. ORDER BY 관련 키워드

키워드	의미	예시
정렬 , 순서 , 내림차순 , 오름차순	전체 결과를 정렬	점수 높은 순 정렬

6. HAVING 절 관련 키워드

키워드	의미	예시
집계 결과 조건 , ~이상인 그룹 , ~이하인 그룹	그룹화된 결과에 조건 적용	주문 수가 10건 이상인 고객

7. DISTINCT 관련 키워드

키워드	의미	예시
중복 제거 , 고유한 , 유일한	중복 데이터 제거	DISTINCT 도시 개수

▼ **DISTINCT 더 알아보기**

DISTINCT 는 **중복된 값을 제거하고 고유한(Unique) 값만 출력**할 때 사용합니다.

✅ **기본 사용법**

```
SELECT DISTINCT column_name
FROM table_name;
```

- 특정 컬럼에서 중복을 제거한 고유 값만 가져옵니다.

✅ **예시 1) 도시 목록 중복 제거**

```
SELECT DISTINCT city
FROM customers;
```

- customers 테이블에 같은 도시가 여러 번 나와도 **중복 없이** 한 번씩만 보여줍니다.
- 예시) 서울, 부산, 대구, 서울, 부산 → 서울, 부산, 대구

✅ **예시 2) 여러 컬럼 조합에서 DISTINCT**

```
SELECT DISTINCT city, country
FROM customers;
```

- city + country 조합이 같은 행은 하나만 출력됩니다.
- 예시) (서울, 한국), (부산, 한국), (서울, 한국) → (서울, 한국), (부산, 한국)

✅ **예제 3: COUNT와 함께 사용**

```
SELECT COUNT(DISTINCT city) AS unique_city_count
FROM customers;
```

- 고객이 사는 **고유 도시 개수**를 구할 수 있습니다.

8. LIMIT / TOP 관련 키워드

키워드	의미	예시
`상위`, `하위`, `몇 개만`	조회 결과 개수 제한	급여 상위 5명

9. CASE / 조건식 관련 키워드

키워드	의미	예시
`조건에 따라`, `분류`, `구간`	조건별로 다른 값 출력	점수에 따라 합격/불합격 표시

10. 서브쿼리 관련 키워드

키워드	의미	예시
`~의 결과를 이용해`, `~에서 뽑은 값으로`	쿼리 안의 쿼리 사용	평균 급여보다 높은 직원 조회

11. 윈도우 함수

키워드	의미	예시
`순위`, `누적`, `이전 값과 비교`	그룹 내 순위, 누적 합계 계산	지점별 매출 순위 (RANK())

3-2. 문제 풀이가 어렵고, 어디서부터 손대야 할지 모를 때

✅ 해결 전략

1. **문제를 2~3개의 질문으로 쪼개기**

   ex) “통과한 사람의 평균 점수?”

   - 통과 기준이 뭐지?
   - 점수는 어디 컬럼?
   - 평균은 group by가 필요한가?
2. **SELECT부터 적기 → FROM → WHERE → GROUP BY → HAVING → ORDER BY**
   - 자주 쓰는 순서로 적어보면 훨씬 정리됩니다.
3. **우선 `SELECT *` 로 해보고 데이터부터 보는 연습을 해보세요!**
4. **SQL 작성 전 체크리스트**

- 문제 정의 (문제 분석) ⇒ **문제의 의도는 무엇일까???**
  - 무엇을 구하고 싶은가 **문장으로 적기**
  - 요청사항을 정확히 아는 것이 중요하다.
  - SQL 작성 전 **문제의 요지 분석**
  - 문제를 보며 예상 쿼리를 구성해보자
    - 예시: 월별로 매출 합계를 구하고 싶다
      - SELECT 써?
      - GROUP BY 써?
      - SUM 써?

5. **데이터 구조 확인**

- 쿼리 쓰기 전 반드시 **테이블의 컬럼명 확인**

4. 문제 풀이

▼ 문제 1

문제 링크: https://school.programmers.co.kr/learn/courses/30/lessons/59044

- 테이블 설명

`ANIMAL_INS` 테이블은 동물 보호소에 들어온 동물의 정보를 담은 테이블입니다. `ANIMAL_INS` 테이블 구조는 다음과 같으며, `ANIMAL_ID`, `ANIMAL_TYPE`, `DATETIME`, `INTAKE_CONDITION`, `NAME`, `SEX_UPON_INTAKE`는 각각 동물의 아이디, 생물 종, 보호 시작일, 보호 시작 시 상태, 이름, 성별 및 중성화 여부를 나타냅니다.

NAME	TYPE	NULLABLE
ANIMAL_ID	VARCHAR(N)	FALSE
ANIMAL_TYPE	VARCHAR(N)	FALSE
DATETIME	DATETIME	FALSE
INTAKE_CONDITION	VARCHAR(N)	FALSE
NAME	VARCHAR(N)	TRUE
SEX_UPON_INTAKE	VARCHAR(N)	FALSE

`ANIMAL_OUTS` 테이블은 동물 보호소에서 입양 보낸 동물의 정보를 담은 테이블입니다. `ANIMAL_OUTS` 테이블 구조는 다음과 같으며,

`ANIMAL_ID`, `ANIMAL_TYPE`, `DATETIME`, `NAME`, `SEX_UPON_OUTCOME`는 각각 동물의 아이디, 생물 종, 입양일, 이름, 성별 및 중성화 여

부를 나타냅니다. `ANIMAL_OUTS` 테이블의 `ANIMAL_ID`는 `ANIMAL_INS`의 `ANIMAL_ID`의 외래 키입니다.

NAME	TYPE	NULLABLE
ANIMAL_ID	VARCHAR(N)	FALSE
ANIMAL_TYPE	VARCHAR(N)	FALSE
DATETIME	DATETIME	FALSE
NAME	VARCHAR(N)	TRUE
SEX_UPON_OUTCOME	VARCHAR(N)	FALSE

- **문제**

아직 **입양을 못 간 동물** 중, **가장 오래** 보호소에 있었던 동물 **3마리**의 이름과 보호 시작일을 조회하는 SQL문을 작성해주세요.

이때 결과는 **보호 시작일 순**으로 조회해야 합니다.

- **예시**

예를 들어, `ANIMAL_INS` 테이블과 `ANIMAL_OUTS` 테이블이 다음과 같다면

`ANIMAL_INS`

ANIMAL_ID	ANIMAL_TYPE	DATETIME	INTAKE_CONDITION	NAME	SEX_UPON_INTAKE
A354597	Cat	2014-05-02 12:16:00	Normal	Ariel	Spayed Female
A373687	Dog	2014-03-20 12:31:00	Normal	Rosie	Spayed Female
A412697	Dog	2016-01-03 16:25:00	Normal	Jackie	Neutered Male
A413789	Dog	2016-04-19 13:28:00	Normal	Benji	Spayed Female
A414198	Dog	2015-01-29 15:01:00	Normal	Shelly	Spayed Female
A368930	Dog	2014-06-08 13:20:00	Normal		Spayed Female

`ANIMAL_OUTS`

ANIMAL_ID	ANIMAL_TYPE	DATETIME	NAME	SEX_UPON_OUTCOME
A354597	Cat	2014-05-02 12:16:00	Ariel	Spayed Female
A373687	Dog	2014-03-20 12:31:00	Rosie	Spayed Female
A368930	Dog	2014-06-13 15:52:00		Spayed Female

SQL문을 실행하면 다음과 같이 나와야 합니다.

NAME	DATETIME
Shelly	2015-01-29 15:01:00
Jackie	2016-01-03 16:25:00
Benji	2016-04-19 13:28:00

※ 입양을 가지 못한 동물이 3마리 이상인 경우만 입력으로 주어집니다.

▼ 문제 풀이 아이디어

❓ **입양(ANIMAL_OUTS)에 기록이 없는 동물 중 보호 시작일(DATETIME)이 가장 오래된 3마리**를 찾자!

1. **입양을 못 간 동물**
   - `ANIMAL_INS`에는 있지만 `ANIMAL_OUTS`에는 없는 동물을 찾아야 합니다.

   → `LEFT JOIN ... WHERE O.ANIMAL_ID IS NULL` 또는 `NOT IN` / `NOT EXISTS` 사용
2. **가장 오래 보호소에 있었던 동물**
   - 보호 시작일(`DATETIME`)이 빠른(오래된) 순서대로 정렬해야 합니다. → `ORDER BY DATETIME ASC`
3. **3마리만**
   - 오래된 순으로 정렬한 뒤 상위 3개만 가져와야 합니다. → `LIMIT 3`

✅ 즉, 문제를 읽을 때 **입양 못 간 동물 필터링 → 보호 시작일 기준 정렬 → 3마리 추출 이러한 단계로 생각해보면 쉽습니다.**

▼ 정답 쿼리

1. `LEFT JOIN` 활용

```
SELECT I.NAME, I.DATETIME
FROM ANIMAL_INS I
LEFT JOIN ANIMAL_OUTS O
  ON I.ANIMAL_ID = O.ANIMAL_ID
WHERE O.ANIMAL_ID IS NULL
ORDER BY I.DATETIME
LIMIT 3;
```

2. `NOT IN` 활용

```
SELECT NAME, DATETIME
FROM ANIMAL_INS
WHERE ANIMAL_ID NOT IN (
    SELECT ANIMAL_ID
    FROM ANIMAL_OUTS
)
ORDER BY DATETIME
LIMIT 3;
```

▼ 쿼리 해석

- `SELECT NAME, DATETIME`
  - 결과로 동물의 이름(NAME)과 보호 시작일(DATETIME)만 보여주겠다.
- `FROM ANIMAL_INS`
  - 기준 테이블은 보호소에 처음 들어온 동물 정보(ANIMAL_INS)이다.
- `WHERE ANIMAL_ID NOT IN (SELECT ANIMAL_ID FROM ANIMAL_OUTS)`
  - `ANIMAL_OUTS` 테이블(입양된 동물 정보) 안에 없는 **ANIMAL_ID**를 가진 동물만 선택한다.
  - 즉, **아직 입양을 가지 못한 동물들만 남긴다.**
  - `WHERE 컬럼 NOT IN (값 목록 또는 서브쿼리)`
    - `컬럼`이 괄호 안의 값들에 **포함되지 않는 경우**만 남긴다.
    - 말 그대로 **제외 조건**을 걸 때 사용한다.
- `ORDER BY DATETIME`
  - 그 동물들을 보호 시작일이 오래된 순서(오름차순)로 정렬한다.
- `LIMIT 3`
  - 정렬된 결과 중에서 **앞의 3마리만 가져온다.**

3. `NOT EXISTS` 활용

```
SELECT I.NAME, I.DATETIME
FROM ANIMAL_INS I
WHERE NOT EXISTS (
   SELECT 1
   FROM ANIMAL_OUTS O
   WHERE I.ANIMAL_ID = O.ANIMAL_ID
)
ORDER BY I.DATETIME
LIMIT 3;
```

▼ 쿼리 해석

- `SELECT I.NAME, I.DATETIME`
  - 결과로 동물의 이름(NAME)과 보호 시작일(DATETIME)을 출력한다.
- `FROM ANIMAL_INS I`
  - 기준 테이블은 `ANIMAL_INS` (보호소에 들어온 동물 정보)이고, 이 테이블을 **I**라는 별칭으로 사용한다.
- `WHERE NOT EXISTS ( ... )`

```
WHERE NOT EXISTS (
   SELECT 1
   FROM ANIMAL_OUTS O
   WHERE I.ANIMAL_ID = O.ANIMAL_ID
)
```

  - 조건: 괄호 안의 서브쿼리가 **존재하지 않을 때만** 그 행을 선택한다.
  - `ANIMAL_OUTS` (입양된 동물 정보)에서 `ANIMAL_INS`의 동물 ID(`I.ANIMAL_ID`)와 같은 ID가 있는지 확인한다.
  - `ANIMAL_OUTS` 테이블에서 `I.ANIMAL_ID`와 같은 ID가 존재하는지 확인한다.
  - 여기서 `SELECT 1`은 실제로 숫자 1을 가져오려는 게 아니라, 조건에 맞는 행이 존재하는지만 확인하기 위한 관용적 표현이다. (값 자체는 쓰이지 않음 → EXISTS/NOT EXISTS는 행이 있냐 없냐만 판단)

- 따라서 `ANIMAL_OUTS`에 같은 `ANIMAL_ID`가 있으면 **EXISTS = TRUE**, 없으면 **EXISTS = FALSE**

꼭 1만 써야 하나요?

- `SELECT 1`, `SELECT *`, `SELECT 'abc'` → 전부 동일하게 동작합니다.
- 1을 쓴 이유는?
  - **간단하고 직관적** → 존재만 확인한다는 의미가 드러남.
  - **불필요한 컬럼 조회 방지** → `SELECT *` 보다는 `1`이 가볍게 보이고, 의도가 명확함.
  - **읽는 사람과 DB엔진 모두에게 깔끔한 신호** → 최적화에도 유리.

- `ORDER BY I.DATETIME`
  - 남은 동물들을 보호 시작일(`DATETIME`) 기준으로 오래된 순서(ASC)로 정렬한다.
- `LIMIT 3`
  - 정렬된 결과 중 **상위 3마리만 출력한다.**

▼ 핵심 문법 & 구문 정리

1. `NOT IN`

```
SELECT *
FROM ANIMAL_INS
WHERE ANIMAL_ID NOT IN (
   SELECT ANIMAL_ID
   FROM ANIMAL_OUTS
);
```

- 특징
  - 서브쿼리 결과를 집합으로 만든 뒤, 그 안에 없는 값만 선택.
  - 직관적이고 짧게 쓸 수 있음.
- 주의사항
  - 서브쿼리 결과에 `NULL`이 하나라도 있으면, 비교 결과가 `UNKNOWN`으로 바뀌어 **원하지 않는 결과가 나올 수 있음.**
  - 따라서 보통 `WHERE ANIMAL_ID IS NOT NULL` 조건을 서브쿼리에 붙여서 사용해야 안전함.

2. `NOT EXISTS`

```
SELECT *
FROM ANIMAL_INS I
WHERE NOT EXISTS (
   SELECT 1
   FROM ANIMAL_OUTS O
   WHERE I.ANIMAL_ID = O.ANIMAL_ID
);
```

- 특징
  - **행 존재 여부 검사 방식**
  - `ANIMAL_INS`의 각 행마다, 서브쿼리에 조건을 만족하는 행이 있는지 확인.
  - 없으면(TRUE), 그 행을 결과에 포함.

- 서브쿼리의 `SELECT` 절에 무엇을 써도 상관없음 (`1`, , 문자열 등).
- 장점
  - `NULL` 값에 영향을 받지 않음.
  - 보통 `NOT IN`보다 안전하고, 실무에서 권장됨.

구분	NOT IN	NOT EXISTS
동작 방식	집합에 없는 값 찾기	행이 존재하지 않으면 TRUE
NULL 처리	NULL 있으면 결과 꼬일 수 있음.	NULL 영향 없음 .
가독성	간단하고 직관적	조금 더 길지만 안전
권장 여부	서브쿼리에 NULL 없을 때 사용하는 것을 권장함.	실무에서 더 자주 권장

▼ 문제 2

문제 링크: https://school.programmers.co.kr/learn/courses/30/lessons/164668

- **문제 설명**

다음은 중고 거래 게시판 정보를 담은 `USED_GOODS_BOARD` 테이블과 중고 거래 게시판 사용자 정

보를 담은 `USED_GOODS_USER` 테이블입니다. `USED_GOODS_BOARD` 테이블은 다음과 같으

며 `BOARD_ID`, `WRITER_ID`, `TITLE`, `CONTENTS`, `PRICE`, `CREATED_DATE`, `STATUS`, `VIEWS`는 게시글 ID, 작성자 ID, 게시글 제목, 게시
내용, 가격, 작성일, 거래상태, 조회수를 의미합니다.

Column name	Type	Nullable
BOARD_ID	VARCHAR(5)	FALSE
WRITER_ID	VARCHAR(50)	FALSE
TITLE	VARCHAR(100)	FALSE
CONTENTS	VARCHAR(1000)	FALSE
PRICE	NUMBER	FALSE
CREATED_DATE	DATE	FALSE
STATUS	VARCHAR(10)	FALSE
VIEWS	NUMBER	FALSE

`USED_GOODS_USER` 테이블은 다음과 같으며 `USER_ID`, `NICKNAME`, `CITY`, `STREET_ADDRESS1`, `STREET_ADDRESS2`, `TLNO`는 각

각 회원 ID, 닉네임, 시, 도로명 주소, 상세 주소, 전화번호를 의미합니다.

Column name	Type	Nullable
USER_ID	VARCHAR(50)	FALSE
NICKNAME	VARCHAR(100)	FALSE
CITY	VARCHAR(100)	FALSE
STREET_ADDRESS1	VARCHAR(100)	FALSE
STREET_ADDRESS2	VARCHAR(100)	TRUE
TLNO	VARCHAR(20)	FALSE

- 문제

`USED_GOODS_BOARD`와 `USED_GOODS_USER` 테이블에서 **완료된 중고 거래의 총금액**이 **70만 원 이상**인 사람의 회원 ID, 닉네임, 총거
금액

을 조회하는 SQL문을 작성해주세요. 결과는 **총거래금액을 기준으로 오름차순 정렬**해주세요.

- 예시

`USED_GOODS_BOARD` 테이블이 다음과 같고

BOARD_ID	WRITER_ID	TITLE	CONTENTS	PRICE	CREATED_DATE	STATUS
B0001	zkzkdh1	캠핑의자	가벼워요 깨끗한 상태입니다. 2개	25000	2022-11-29	SALE
B0002	miyeon89	벽걸이 에어컨	엘지 휘센 7평	100000	2022-11-29	SALE
B0003	dhfkzmf09	에어팟 맥스	에어팟 맥스 스카이 블루 색상 판매합니다.	450000	2022-11-26	DONE
B0004	sangjune1	파파야나인 포르쉐 푸쉬카	예민하신분은 피해주세요	30000	2022-11-30	DONE
B0005	zkzkdh1	애플워치7	애플워치7 실버 스텐 45미리 판매합니다.	700000	2022-11-30	DONE

`USED_GOODS_USER` 테이블이 다음과 같을 때

USER_ID	NICKNAME	CITY	STREET_ADDRESS1	STREET_ADDRESS2	TLNO
cjfwls91	점심만금식	성남시	분당구 내정로 185	501호	01036344964
zkzkdh1	후후후	성남시	분당구 내정로 35	가동 1202호	01032777543
spdlqj12	크크큭	성남시	분당구 수내로 206	2019동 801호	01087234922
xlqpfh2	잉여킹	성남시	분당구 수내로 1	001-004	01064534911
dhfkzmf09	찐찐	성남시	분당구 수내로 13	A동 1107호	01053422914

SQL을 실행하면 다음과 같이 출력되어야 합니다.

USER_ID	NICKNAME	TOTAL_SALES
zkzkdh1	후후후	700000

▼ 문제 풀이 아이디어

❓ **중고 거래 완료 건**(`STATUS = 'DONE'`)의 금액을 합산하여, 그 합이 **70만 원 이상인 사용자**를 조회하자!

이 문제에서 가장 먼저 생각해야하는 것은?

🧐 ✅ **중고 거래 완료 건 (`STATUS = 'DONE'`) 조건**

- 문제는 "완료된 중고 거래의 총금액"을 묻고 있어요.
- 따라서 **모든 게시글**이 아니라, **거래 완료된 게시글만** 대상으로 해야 합니다.
- 즉, `USED_GOODS_BOARD` 테이블에서 `STATUS = 'DONE'` 조건으로 **필터링**하는 게 출발점입니다.

1. **거래 완료된 건만**
   - USED_GOODS_BOARD에서 `STATUS = 'DONE'` 조건으로 필터링
   - 완료된 거래 금액만 합산해야 함.
2. **회원별 총 거래 금액**
   - `WRITER_ID` 기준으로 `SUM(PRICE)` 집계
   - 회원 정보(NICKNAME 등)를 가져오기 위해 `USED_GOODS_USER`와 조인
3. **70만 원 이상**
   - 집계 결과 중 `SUM(PRICE) >= 700000` 조건을 `HAVING` 절로 설정
4. **정렬**
   - 총 거래 금액(`TOTAL_SALES`) 기준으로 오름차순 정렬

✅ 즉, 문제를 읽을 때 **조건 필터링 → 집계 → 조인 → 집계 조건 필터링 → 정렬** 이러한 단계로 풀어보겠다고 먼저 생각해보면 좋아요.

▼ 정답 쿼리

1. JOIN + GROUP BY + HAVING 활용

- JOIN → WHERE → GROUP BY → HAVING (한 번에 처리)

```
SELECT
   U.USER_ID,
   U.NICKNAME,
   SUM(B.PRICE) AS TOTAL_SALES
```

```
FROM USED_GOODS_BOARD B
JOIN USED_GOODS_USER U
   ON B.WRITER_ID = U.USER_ID
WHERE B.STATUS = 'DONE'
GROUP BY U.USER_ID, U.NICKNAME
HAVING SUM(B.PRICE) >= 700000
ORDER BY TOTAL_SALES ASC;
```

▼ 쿼리 해석

1. **FROM**

```
FROM USED_GOODS_BOARD B
JOIN USED_GOODS_USER U
   ON B.WRITER_ID = U.USER_ID
```

- 먼저 `USED_GOODS_BOARD`와 `USED_GOODS_USER`를 조인합니다.
- 결과: 모든 게시글에 대해 작성자의 회원 정보가 붙은 "결합된 데이터셋" 생성.

2. **WHERE**

```
WHERE B.STATUS = 'DONE'
```

- 조인된 결과에서 `STATUS = 'DONE'` 조건을 만족하는 행만 남김.
- 즉, 거래 완료된 데이터만 필터링.

3. **GROUP BY**

```
GROUP BY U.USER_ID, U.NICKNAME
```

- 남은 데이터들을 **회원별**(`USER_ID`, `NICKNAME`)로 묶습니다.
- 한 회원이 여러 건의 거래를 했다면 하나의 그룹으로 합쳐짐.

4. **HAVING**

```
HAVING SUM(B.PRICE) >= 700000
```

- 그룹별 집계 결과(`SUM(B.PRICE)`)를 계산한 뒤,
  총합이 70만 원 이상인 그룹만 남깁니다.
- 주의: `WHERE`는 집계 이전 필터링, `HAVING`은 집계 이후 필터링 (여기서 `WHERE`가 아닌 `HAVING`을 쓰는 이유는 **집계 결과에 조건을 거는 절**이기 때문임.)

5. **SELECT**

```
SELECT
   U.USER_ID,
   U.NICKNAME,
   SUM(B.PRICE) AS TOTAL_SALES
```

- 각 그룹에서 보여줄 컬럼만 선택.
- 회원 ID, 닉네임, 총 거래금액(`SUM(B.PRICE)`) 출력.

6. **ORDER BY**

```
ORDER BY TOTAL_SALES ASC;
```

- 마지막으로 출력된 결과를 **총 거래 금액 기준 오름차순**으로 정렬.

2. 서브쿼리 활용

- 서브쿼리에서 집계 끝낸 뒤 → USER 테이블과 JOIN (단계 분리)

```
SELECT
   U.USER_ID,
   U.NICKNAME,
   S.TOTAL_SALES
FROM USED_GOODS_USER U
JOIN (
   SELECT WRITER_ID, SUM(PRICE) AS TOTAL_SALES
   FROM USED_GOODS_BOARD
   WHERE STATUS = 'DONE'
   GROUP BY WRITER_ID
   HAVING SUM(PRICE) >= 700000
) S
ON U.USER_ID = S.WRITER_ID
ORDER BY S.TOTAL_SALES ASC;
```

▼ 쿼리 해석

1. 내부 서브쿼리 (S)

```
SELECT WRITER_ID, SUM(PRICE) AS TOTAL_SALES
FROM USED_GOODS_BOARD
WHERE STATUS = 'DONE'
GROUP BY WRITER_ID
HAVING SUM(PRICE) >= 700000
```

- `USED_GOODS_BOARD`에서 거래 상태가 `DONE`인 데이터만 선택.
- `WRITER_ID` 기준으로 묶어(`GROUP BY`) 각 회원이 완료된 거래에서 판매한 금액 합계를 계산.
- `SUM(PRICE)`가 **70만 원 이상**인 회원만 남김.
- 결과

WRITER_ID	TOTAL_SALES
zkzkdh1	700000

2. 외부 쿼리 (U + S JOIN)

```
FROM USED_GOODS_USER U
JOIN ( ... ) S
ON U.USER_ID = S.WRITER_ID
```

- `USED_GOODS_USER` 테이블과 서브쿼리 결과 `S`를 조인.
- `USER_ID = WRITER_ID` 조건으로 매칭하여, **회원 정보(닉네임 등) + 총 거래 금액**을 한 행으로 묶음.

3. SELECT

```
SELECT
    U.USER_ID,
    U.NICKNAME,
    S.TOTAL_SALES
```

- 최종적으로 출력할 컬럼은
  - `USER_ID` : 회원 ID
  - `NICKNAME` : 회원 닉네임
  - `TOTAL_SALES` : 총 거래 금액

4. ORDER BY

```
ORDER BY S.TOTAL_SALES ASC;
```

- 총 거래 금액 기준으로 오름차순 정렬.

▼ 이 문제를 서브쿼리로 풀 수 있는 이유는?

1. **집계 결과를 독립적으로 만들 수 있음**
   - 우리가 필요한 건 회원별 거래 완료 금액 합계예요.
   - 이 합계는 `USED_GOODS_BOARD` 테이블만으로 계산할 수 있죠.

     ```
     SELECT WRITER_ID, SUM(PRICE) AS TOTAL_SALES
     FROM USED_GOODS_BOARD
     WHERE STATUS = 'DONE'
     GROUP BY WRITER_ID
     ```

   - 이렇게 만든 결과는 하나의 임시 테이블(뷰)처럼 다룰 수 있습니다.
2. **조건(HAVING)으로 필요한 데이터만 걸러낼 수 있음**
   - 합계가 **70만 원 이상**인 사용자만 필요하기 때문에, 서브쿼리 안에서 이미 걸러낼 수 있습니다.

     ```
     HAVING SUM(PRICE) >= 700000
     ```

3. **서브쿼리 결과를 다른 테이블과 조인할 수 있음**
   - 서브쿼리 결과는 `WRITER_ID, TOTAL_SALES`만 갖고 있어요.
   - 그런데 문제에서는 **닉네임(NICKNAME)** 같은 추가 정보도 필요합니다.
   - 따라서, 외부 쿼리에서 `USED_GOODS_USER` 테이블과 `JOIN`하여 보강할 수 있습니다.
4. **단계 분리 → 가독성 및 유지보수성 향상**
   - 총 거래금액 계산과 회원 정보 붙이기라는 두 단계를 **논리적으로 분리**할 수 있어서 쿼리 구조가 더 명확해질 수 있습니다.
   - 실무에서는 복잡한 조건이 많을 때 이 방식이 선호됩니다.

🤓 **회원별 총합**이라는 집계 결과를 별도로 만든 뒤, 이를 다시 **회원 정보**와 조인해야 하므로 서브쿼리를 쓸 수 있다.

3. WITH 구문(CTE 활용)

```
WITH SALES AS (
   SELECT WRITER_ID, SUM(PRICE) AS TOTAL_SALES
   FROM USED_GOODS_BOARD
   WHERE STATUS = 'DONE'
   GROUP BY WRITER_ID
)
SELECT
   U.USER_ID,
   U.NICKNAME,
   S.TOTAL_SALES
FROM SALES S
JOIN USED_GOODS_USER U
  ON U.USER_ID = S.WRITER_ID
WHERE S.TOTAL_SALES >= 700000
ORDER BY S.TOTAL_SALES ASC;
```

▼ 쿼리 해석

1. CTE 정의 (`SALES`)

```
WITH SALES AS (
   SELECT WRITER_ID, SUM(PRICE) AS TOTAL_SALES
   FROM USED_GOODS_BOARD
   WHERE STATUS = 'DONE'
   GROUP BY WRITER_ID
)
```

- `USED_GOODS_BOARD` 테이블에서 거래 상태가 `DONE` 인 데이터만 선택.
- `WRITER_ID` 기준으로 그룹화하여(`GROUP BY`) 각 회원의 총 거래 금액(`SUM(PRICE)`)을 계산.
- 결과는 `SALES` 라는 이름의 임시 테이블로 저장됨.

WRITER_ID	TOTAL_SALES
zkzkdh1	700000
dhfkzmf09	450000

2. 외부 SELECT

```
SELECT
   U.USER_ID,
   U.NICKNAME,
   S.TOTAL_SALES
FROM SALES S
JOIN USED_GOODS_USER U
  ON U.USER_ID = S.WRITER_ID
```

- CTE `SALES` (회원별 총 거래금액)와 `USED_GOODS_USER` (회원 정보)를 `USER_ID = WRITER_ID` 로 조인.
- 즉, **회원 ID, 닉네임, 총 거래금액**을 하나의 결과로 묶음.

3. WHERE (집계 이후 조건)

```
WHERE S.TOTAL_SALES >= 700000
```

- `TOTAL_SALES`가 70만 원 이상인 회원만 남김.
- 집계는 이미 CTE 안에서 끝났기 때문에, 여기서는 `WHERE` 절을 사용해도 됨. (서브쿼리/CTE 밖에서는 `HAVING` 대신 `WHERE`로 필터링 가능)

4. ORDER BY

```
ORDER BY S.TOTAL_SALES ASC;
```

- 최종 결과를 총 거래 금액 기준으로 오름차순 정렬

▼ 이 문제를 WITH 구문(CTE 활용)으로 풀 수 있는 이유는?

1. **중간 결과(집계)를 재사용 가능**
   - 문제 해결의 핵심은 회원별 완료 거래 금액 합계를 먼저 구하는 것!!
   - 이 결과는 `USED_GOODS_BOARD` 테이블만으로 만들 수 있고, CTE를 이용하면 이 집계 결과를 **하나의 임시 테이블**처럼 정의할 수 있음.

```
WITH SALES AS (
    SELECT WRITER_ID, SUM(PRICE) AS TOTAL_SALES
    FROM USED_GOODS_BOARD
    WHERE STATUS = 'DONE'
    GROUP BY WRITER_ID
)
```

   → `SALES`라는 가상의 테이블 생성.

2. **집계 후 필터링을 WHERE로 간단히 처리 가능**
   - CTE 안에서는 단순히 합계를 구하고, 바깥 쿼리에서는 `WHERE TOTAL_SALES >= 700000` 조건을 걸 수 있음.
   - 즉, `HAVING`이 아니라 **WHERE 절로 필터링 가능** → 쿼리의 가독성 향상

3. **쿼리 구조를 단계적으로 나눌 수 있음**
   - CTE를 쓰면 쿼리를 "두 단계"로 나눌 수 있습니다.
     1. **집계 단계**: 회원별 거래금액 합계 계산
     2. **출력 단계**: 회원 정보와 조인 후 필터링 및 정렬
   - 이렇게 하면 쿼리 읽는 사람이 **데이터 흐름을 직관적으로 이해**할 수 있음.

1. **복잡한 쿼리 확장에 유리**
   - 만약 조건이 더 많아지거나, 예를 들어 "도시별 합계, 상위 N명" 같은 추가 요구사항이 생겨도 이미 만들어둔 `SALES` CTE를 **재사용**하면 쉽게 확장 가능.

▼ 문제 3

문제 링크: https://school.programmers.co.kr/learn/courses/30/lessons/59045

- **문제 설명**

`ANIMAL_INS` 테이블은 동물 보호소에 들어온 동물의 정보를 담은 테이블입니다. `ANIMAL_INS` 테이블 구조는 다음과 같으며, `ANIMAL_ID`, `ANIMAL_TYPE`, `DATETIME`, `INTAKE_CONDITION`, `NAME`, `SEX_UPON_INTAKE`는 각각 동물의 아이디, 생물 종, 보호 시작일, 보호 시작 시 상태, 이름, 성별 및 중성화 여부를 나타냅니다.

NAME	TYPE	NULLABLE
ANIMAL_ID	VARCHAR(N)	FALSE
ANIMAL_TYPE	VARCHAR(N)	FALSE
DATETIME	DATETIME	FALSE
INTAKE_CONDITION	VARCHAR(N)	FALSE
NAME	VARCHAR(N)	TRUE
SEX_UPON_INTAKE	VARCHAR(N)	FALSE

`ANIMAL_OUTS` 테이블은 동물 보호소에서 입양 보낸 동물의 정보를 담은 테이블입니다. `ANIMAL_OUTS` 테이블 구조는 다음과 같으며, `ANIMAL_ID`, `ANIMAL_TYPE`, `DATETIME`, `NAME`, `SEX_UPON_OUTCOME`는 각각 동물의 아이디, 생물 종, 입양일, 이름, 성별 및 중성화 여부를 나타냅니다. `ANIMAL_OUTS` 테이블의 `ANIMAL_ID`는 `ANIMAL_INS`의 `ANIMAL_ID`의 외래키입니다.

NAME	TYPE	NULLABLE
ANIMAL_ID	VARCHAR(N)	FALSE
ANIMAL_TYPE	VARCHAR(N)	FALSE
DATETIME	DATETIME	FALSE
NAME	VARCHAR(N)	TRUE
SEX_UPON_OUTCOME	VARCHAR(N)	FALSE

- 문제

🗣️ 보호소에서 중성화 수술을 거친 동물 정보를 알아보려 합니다. 보호소에 들어올 **당시에는 중성화되지 않았지만**, 보호소를 **나갈 당시에는 중성화된 동물의 아이디와 생물 종, 이름을 조회하는 아이디 순으로 조회**하는 SQL 문을 작성해주세요.

- 예시

예를 들어, `ANIMAL_INS` 테이블과 `ANIMAL_OUTS` 테이블이 다음과 같다면

`ANIMAL_INS`

ANIMAL_ID	ANIMAL_TYPE	DATETIME	INTAKE_CONDITION	NAME	SEX_UPON_INTAKE
A367438	Dog	2015-09-10 16:01:00	Normal	Cookie	Spayed Female
A382192	Dog	2015-03-13 13:14:00	Normal	Maxwell 2	Intact Male
A405494	Dog	2014-05-16 14:17:00	Normal	Kaila	Spayed Female
A410330	Dog	2016-09-11 14:09:00	Sick	Chewy	Intact Female

`ANIMAL_OUTS`

ANIMAL_ID	ANIMAL_TYPE	DATETIME	NAME	SEX_UPON_OUTCOME
A367438	Dog	2015-09-12 13:30:00	Cookie	Spayed Female
A382192	Dog	2015-03-16 13:46:00	Maxwell 2	Neutered Male

ANIMAL_ID	ANIMAL_TYPE	DATETIME	NAME	SEX_UPON_OUTCOME
A405494	Dog	2014-05-20 11:44:00	Kaila	Spayed Female
A410330	Dog	2016-09-13 13:46:00	Chewy	Spayed Female

- Cookie는 보호소에 들어올 당시에 이미 중성화되어있었습니다.
- Maxwell 2는 보호소에 들어온 후 중성화되었습니다.
- Kaila는 보호소에 들어올 당시에 이미 중성화되어있었습니다.
- Chewy는 보호소에 들어온 후 중성화되었습니다.

따라서 SQL문을 실행하면 다음과 같이 나와야 합니다.

ANIMAL_ID	ANIMAL_TYPE	NAME
A382192	Dog	Maxwell 2
A410330	Dog	Chewy

▼ 문제 풀이 아이디어

이 문제에서 가장 먼저 생각해야하는 것은?

🤓 ✅ **비교 대상은 무엇인가?**

- 문제에서 찾고 싶은 건 **"보호소에 들어올 때 상태"와 "나갈 때 상태"의 차이**예요.
- 즉, 한 테이블만 보는 게 아니라, `ANIMAL_INS` **(입소 시 정보)** 와 `ANIMAL_OUTS` (출소 시 정보)를 **비교**해야 합니다.

1. **두 테이블을 연결해야 한다**
   - 두 테이블에 공통 키(`ANIMAL_ID`)가 있으니, `JOIN`이 필요하다는 걸 먼저 생각해야 합니다.
2. **비교할 속성은 무엇인가?**
   - 단순히 날짜나 상태가 아니라, 이 문제에서 중요한 건 성별 및 중성화 여부(`SEX_UPON_INTAKE`, `SEX_UPON_OUTCOME`)입니다.
3. **조건 차이를 찾는 문제라는 것**
   - 입소 시 → `Intact` (중성화 X)
   - 출소 시 → `Spayed` 또는 `Neutered` (중성화 O)
   - 이 조건 차이를 SQL에서 `WHERE`로 구현해야 합니다.

▼ 정답 쿼리

잊지 말자 문제 조건!

- `SEX_UPON_INTAKE` 컬럼이 `"Intact"`를 포함해야 함.

🤓
- `SEX_UPON_OUTCOME` 컬럼이 `"Neutered"` 또는 `"Spayed"`를 포함해야 함.
- 두 테이블을 `ANIMAL_ID`로 JOIN.
- 결과는 `ANIMAL_ID` 기준 오름차순.

1. **JOIN 활용**

```
SELECT i.ANIMAL_ID,
       i.ANIMAL_TYPE,
       i.NAME
FROM ANIMAL_INS i
JOIN ANIMAL_OUTS o
  ON i.ANIMAL_ID = o.ANIMAL_ID
WHERE i.SEX_UPON_INTAKE LIKE 'Intact%'
  AND (o.SEX_UPON_OUTCOME LIKE 'Spayed%' OR o.SEX_UPON_OUTCOME LIKE 'Neutered%')
ORDER BY i.ANIMAL_ID;
```

▼ 쿼리 해석

1. SELECT 절

```
SELECT i.ANIMAL_ID,
       i.ANIMAL_TYPE,
       i.NAME
```

- 결과로 보여줄 컬럼을 선택합니다.
- `ANIMAL_INS` 테이블의 **동물 ID, 동물 종류, 이름**만 출력

2. FROM + JOIN 절

```
FROM ANIMAL_INS i
JOIN ANIMAL_OUTS o
  ON i.ANIMAL_ID = o.ANIMAL_ID
```

- `ANIMAL_INS` 테이블을 `i` 라는 별칭으로,
  `ANIMAL_OUTS` 테이블을 `o` 라는 별칭으로 사용.
- 두 테이블을 **동물 ID(`ANIMAL_ID`)** 기준으로 `JOIN` → 같은 동물의 입소 정보와 출소 정보를 연결

3. **WHERE 절 (핵심)**

```
WHERE i.SEX_UPON_INTAKE LIKE 'Intact%'
  AND (o.SEX_UPON_OUTCOME LIKE 'Spayed%' OR o.SEX_UPON_OUTCOME LIKE 'Neutered%')
```

- 조건절
  1. 입소 시 상태(`SEX_UPON_INTAKE`)가 `"Intact..."` → 중성화 안 된 상태
  2. 출소 시 상태(`SEX_UPON_OUTCOME`)가 `"Spayed..."` 또는 `"Neutered..."` → 중성화 완료 상태

- `LIKE 'Intact%'`
  - `Intact`로 **시작**하고 뒤에 어떤 글자든 올 수 있다는 뜻(`%`는 0글자 이상 와일드카드)
  - `Intact Male`, `Intact Female` 모두 매칭
- `LIKE 'Spayed%' OR LIKE 'Neutered%'`
  - `Spayed ...` 또는 `Neutered ...`로 시작하는 문자열만 선별
  - 성별이 무엇이든 상관없이 **중성화된 상태**만 잡아냄.

- 즉, 보호소에 들어올 때는 중성화 X, 나갈 때는 중성화 O인 동물만 선택

4. **ORDER BY 절**

```
ORDER BY i.ANIMAL_ID;
```

- 결과를 `ANIMAL_ID` 기준으로 오름차순 정렬

2. **CASE WHEN 사용**

```
SELECT i.ANIMAL_ID,
       i.ANIMAL_TYPE,
       i.NAME
FROM ANIMAL_INS i
JOIN ANIMAL_OUTS o
  ON i.ANIMAL_ID = o.ANIMAL_ID
WHERE CASE
        WHEN i.SEX_UPON_INTAKE LIKE '%Intact%' THEN 'N'
        ELSE 'Y'
      END = 'N'
  AND CASE
        WHEN o.SEX_UPON_OUTCOME LIKE '%Intact%' THEN 'N'
        ELSE 'Y'
      END = 'Y'
ORDER BY i.ANIMAL_ID;
```

▼ 쿼리 해석

🤓 첫 번째 쿼리(`LIKE 'Intact%'` 등 직접 비교)와 결과는 동일하지만, 이 쿼리는 `CASE` 문으로 **중성화 여부를 Y/N으로 변환**해서 조건을 직관적으로 표현한 방식!

1. SELECT 절

```
SELECT i.ANIMAL_ID,
       i.ANIMAL_TYPE,
       i.NAME
```

- 결과로 출력할 컬럼을 지정합니다.
- `ANIMAL_INS` 테이블(`i`)에서 **동물의 ID, 종류, 이름**을 가져옵니다.

2. FROM + JOIN 절

```
FROM ANIMAL_INS i
JOIN ANIMAL_OUTS o
  ON i.ANIMAL_ID = o.ANIMAL_ID
```

- `ANIMAL_INS` 테이블(`i`)과 `ANIMAL_OUTS` 테이블(`o`)을 **동물 ID(ANIMAL_ID)** 기준으로 INNER JOIN 합니다.

- 같은 동물의 **입소 정보와 출소 정보**를 한 행에 합칩니다.

3. WHERE 절 (핵심)

```
WHERE CASE
        WHEN i.SEX_UPON_INTAKE LIKE '%Intact%' THEN 'N'
        ELSE 'Y'
      END = 'N'
  AND CASE
        WHEN o.SEX_UPON_OUTCOME LIKE '%Intact%' THEN 'N'
        ELSE 'Y'
      END = 'Y'
```

- 입소 시 상태 확인
  - `i.SEX_UPON_INTAKE`가 `"Intact..."` → `'N'` (Not neutered, 중성화 X)
  - 그 외(`Spayed Female`, `Neutered Male` 등) → `'Y'` (중성화 O)
  - 조건: `= 'N'` → 입소할 때 **중성화되지 않은 동물**만 남김.
- 출소 시 상태 확인
  - `o.SEX_UPON_OUTCOME`가 `"Intact..."` → `'N'` (중성화 X)
  - 그 외(`Spayed...`, `Neutered...`) → `'Y'` (중성화 O)
  - 조건: `= 'Y'` → 출소할 때 **중성화된 동물**만 남김.

따라서, **입소 당시엔 중성화 X(`N`), 출소 당시엔 중성화 O(`Y`)인 동물만 필터링!**

4. **ORDER BY 절**

```
ORDER BY i.ANIMAL_ID;
```

- 최종 결과를 동물의 ID 기준으로 오름차순 정렬합니다.