[베이직반] SQL 1회차

구분	선택 학습반
날짜	@2025년 9월 25일 오후 7:00
대상	베이직
태그	SQL
튜터	강 강민정

[강의 녹화본]

[강의 자료]

📖 목차

1. SQL의 작동순서 및 작성순서

1-1. 작성 순서

- SQL의 문법이죠! 작성순서는 아래와 같아요.
- 제발 외웁시다.

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

1-2. 작동순서

- SQL이 내부적으로 인지하고 작동하는 순서는 아래와 같습니다.
    - 테이블을 확인한 후, 조인 조건을 확인하고, 조인 후 추출 조건을 확인해요.
    - 그 뒤로 데이터 그룹화, 추출, 중복제거 후 마지막으로 데이터를 정렬해요.
- SQL에 익숙해지면 당연한 원리이지만 일단 외우는 것을 권장합니다.

🤓 **FROM → ON → JOIN → WHERE → GROUP BY → 집계함수 → HAVING → SELECT → DISTINCT → ORDER BY → LIMIT**

작동 순서 상세하게 알아보기

1️⃣ FROM

- 어떤 테이블(데이터 원본)에서 가져올지 결정

2️⃣ JOIN / ON

- 여러 테이블을 결합하고, ON 조건에 따라 행을 매칭
- 여러 테이블 JOIN 시 먼저 두 개만 조인하여 결과를 확인!
  - 조인 조건이 잘못되면 데이터가 폭발적으로 늘어나거나 없어질 수 있음.
- JOIN 시 NULL 값이 필요한 경우가 있으니 JOIN 방식 고려하기

3️⃣ WHERE

- 결합된 데이터에서 **조건에 맞는 행만 필터링**
- WHERE 필터링을 적용할 때 NULL도 제외되는 것을 기억하기

4️⃣ GROUP BY

5️⃣ 집계함수

☝️

- 그룹별 집계 (SUM, AVG, COUNT ...)

6️⃣ HAVING

- 그룹화된 집계 결과에 조건 적용 (`HAVING`은 집계된 결과를 필터링 함.)
- 예시: HAVING SUM(sales) > 100

7️⃣ SELECT

- 최종적으로 출력할 컬럼 / 표현식을 선택
- 집계 함수(SUM, COUNT, AVG 등)도 이 시점에서 계산됨

8️⃣ DISTINCT

- SELECT 결과에서 중복 행 제거

9️⃣ ORDER BY

- 결과를 정렬

🔟 LIMIT

- 결과에서 일부만 가져오기 (예: 상위 10개)

- **작동순서를 꼭 이해하고 가야하는 이유는? 🤔**
  - **SQL 구문작성 후 에러가 발생했을 경우, 문제를 해결하는 데 시간이 오래 걸리기 때문입니다.**
  - 복잡한 쿼리를 분석할 수 있어요. (특히 `JOIN`, `GROUP BY`, `HAVING` 등이 섞일 때 이해가 쉬움)
  - SQL 작동 순서를 이해하면, 에러 해결, 쿼리 설계, 성능 개선이 쉬워집니다.

1-3. 예시로 생각해보기

1. 아래 두 개의 테이블이 있다고 가정합시다.

- `STUDENT` (학생 정보: `STUDENT_ID`, `NAME`, `DEPT_ID`)

STUDENT_ID	NAME	DEPT_ID
1	철수	10
2	영희	20
3	민수	10
4	나래	30

- `GRADE` (성적 정보: `STUDENT_ID`, `SUBJECT`, `SCORE`)

STUDENT_ID	SUBJECT	SCORE
1	SQL	80
2	SQL	60
3	SQL	90
4	SQL	70

예시) “과목이 SQL인 경우, 부서별 평균 점수를 구해서 평균이 80점 이상인 학과만 출력한다. 그 결과를 평균 점수가 높은 순으로 정렬해서 상위 2개만 본다.”

- 쿼리 작성

```
SELECT S.DEPT_ID, AVG(G.SCORE) AS AVG_SCORE   -- SELECT 6
FROM STUDENT S                                -- FROM 1
JOIN GRADE G ON S.STUDENT_ID = G.STUDENT_ID   -- JOIN / ON 2
WHERE G.SUBJECT = 'SQL'                       -- WHERE 3
GROUP BY S.DEPT_ID                            -- GROUP BY 4
HAVING AVG(G.SCORE) >= 80                     -- HAVING 5
ORDER BY AVG_SCORE DESC                       -- ORDER BY 7
LIMIT 2;                                      -- LIMIT 8
```

▼ 쿼리 실행 흐름 (작동순서)

1) FROM → STUDENT

- 먼저 학생 테이블을 가져옵니다.

STUDENT_ID	NAME	DEPT_ID
1	철수	10
2	영희	20
3	민수	10
4	나래	30

2) JOIN / ON → GRADE 결합

- 학번으로 성적 테이블을 붙입니다.

STUDENT_ID	NAME	DEPT_ID	SUBJECT	SCORE
1	철수	10	SQL	80
2	영희	20	SQL	60
3	민수	10	SQL	90
4	나래	30	SQL	70

3) WHERE G.SUBJECT = 'SQL'

- 이번 예시에선 전부 SQL이라 **변화 없음.** (다른 과목이 있었다면 이 단계에서 제거됩니다.)

4) GROUP BY S.DEPT_ID

- 부서별로 묶입니다.
  - DEPT_ID=10 → [80, 90]
  - DEPT_ID=20 → [60]
  - DEPT_ID=30 → [70]

5) 집계 함수(AVG)

- 각 그룹의 평균을 계산합니다.

DEPT_ID	AVG_SCORE
10	85
20	60
30	70

6) HAVING AVG(G.SCORE) ≥ 80

- 평균이 80 이상인 그룹만 남깁니다.

DEPT_ID	AVG_SCORE
10	85

7) SELECT

- 필요한 컬럼만 남깁니다. (`DEPT_ID` , `AVG_SCORE`)

8) ORDER BY AVG_SCORE DESC

- 1개 행뿐이라 순서는 동일합니다. (여러 행이면 평균 내림차순 정렬)

9) LIMIT 1

- 상위 1개만 출력
- 최종 결과

DEPT_ID	AVG_SCORE
10	85

✅ **쿼리 실행 흐름 (작동순서) 정리**

- **FROM/JOIN**으로 "큰 중간 테이블"을 만들고
- **WHERE**로 행을 먼저 걸러낸 다음
- **GROUP BY + 집계**로 그룹 단위 값을 만든 뒤
- **HAVING**으로 그룹 결과를 다시 걸러
- **SELECT → ORDER BY → LIMIT**로 최종 모양을 만든다.

2. GROUP BY

✅ GROUP BY

- `GROUP BY`절은 앞에서 배운 집계함수에 그룹(기준)이 더해진 개념이예요.
- 전체 데이터를 기준으로 조회할 때는 `GROUP BY`절이 필요하지 않지만, 특정 컬럼을 기준으로 데이터를 요약해서 비교하고 싶을 때 주로 사용해요.
  - **GROUP BY 없이**: 전체 데이터를 하나의 그룹으로 간주 → 전체 합계, 전체 평균 등 전체 집계
  - **GROUP BY 사용**: 특정 컬럼(나이 등)을 기준으로 나눠서 각 그룹의 집계 결과를 비교하고 싶을 때
- SQL `GROUP BY` 기본 작성법은 아래와 같아요.

```
SELECT
  기준컬럼,
  집계함수1(조건컬럼) AS 별칭1,
  집계함수2(조건컬럼) AS 별칭2
FROM 테이블명
WHERE 조건    -- 선택적으로 사용 가능
GROUP BY 기준컬럼
;
```

1. **GROUP BY + 집계 함수**를 쓸 때 지켜야 할 핵심 규칙
   - SELECT 문 뒤 기준 컬럼(나라, 성별, 레벨..등등), 집계함수(`COUNT`, `MAX`, `MIN`, `AVG`, `SUM`) 작성
   - WHERE 절 뒤 `GROUP BY` 기준컬럼 작성 (WHERE 절은 생략 가능합니다.)
   - `SELECT` 문에 있는 컬럼은 `GROUP BY`에 있어야 한다. (컬럼명끼리 동일, 표현식이면 표현식)

🍀 2. **작동순서 보면서 생각해보기!**

- **FROM → ON → JOIN → WHERE → GROUP BY → 집계함수 → HAVING → SELECT → DISTINCT → ORDER BY → LIMIT**
  - `GROUP BY`는 SQL 실행 순서상 `WHERE` 다음, `SELECT`보다 먼저 실행됩니다.
  - 따라서 데이터를 그룹화한 뒤 `SELECT`가 실행되기 때문에, `SELECT` 절에는 그룹 기준 컬럼이나 집계 함수로 계산된 값만 쓸 수 있습니다. (SELECT에는 `집계 함수` or `GROUP BY에 있는 컬럼` 만 올 수 있어요.)
  - `GROUP BY` 에 포함되지 않은 일반 컬럼을 그냥 쓰면, 그 값이 어느 그룹의 데이터인지 명확하지 않아서 에러가 발생합니다.

▼ 이게 무슨 말이냐고요? 조금 더 쉽게 생각해봅시다! 🤔

✅ 예를 들어, 직원 테이블(`employees`)이 있다고 해볼게요.

department	name	salary
Sales	Kim	3000
Sales	Lee	2500
IT	Park	4000
IT	Choi	4200

여기서 <부서 **별** 평균 월급>을 구하고 싶으면, `GROUP BY department` 라고 하면 됩니다.

그러면 SQL은 같은 `department`끼리 모아서 그룹을 만들어요.

✅ **GROUP BY 결과에는 어떤 값을 SELECT 할 수 있을까?**

- 그룹을 만들면, 각 그룹을 대표할 수 있는 값만 결과에 남아요. 그래서 **SELECT 절에는 두 가지 종류가 올 수 있습니다!**

1. **그룹 기준으로 묶은 컬럼** (`GROUP BY`에 적힌 것)

   → `department`

2. **집계 함수**

   → `AVG(salary)`, `SUM(salary)`, `COUNT(*)`

✅ **잘못된 예시 (에러 발생)**

```
SELECT department, name, AVG(salary)
FROM employees
GROUP BY department;
```

- `department` → 그룹 기준이니까 OK
- `AVG(salary)` → 집계 함수니까 OK
  - 왜 집계 함수는 될까요?
    - 집계 함수(AVG, SUM, COUNT, MIN, MAX ...)는 **그룹 안의 여러 값을 하나로 줄여주는 기능**이에요.
    - 즉, **모호성을 없애주는 도구**라고 볼 수 있습니다.
    - **집계 함수**: 여러 값을 하나의 값으로 만들어 주므로 에러 없음.
    - **일반 컬럼**: 그룹 안에 값이 여러 개일 수 있는데, 줄이는 규칙이 없으니 에러 발생!
- `name` → 문제 발생!
- **왜 문제일까?**
  - `department`별로 그룹을 만들었는데, 그 안에는 여러 명의 `name`이 있어요.
  - 예를 들어, Sales 부서에는 Kim과 Lee가 있는데 **"어떤 이름을 출력해야 하지?"** 라는 모호함이 생깁니다.
  - SQL은 이걸 스스로 결정할 수 없으니 에러를 내는 거예요.

✅ **해결 방법**

1. `name`도 그룹 기준에 포함시키기

```
SELECT department, name, AVG(salary)
FROM employees
GROUP BY department, name;
```

- <부서 + 이름> 단위로 그룹을 묶습니다.
- 즉, Sales 부서의 Kim과 Lee가 각각 따로 계산돼요.

2. 집계 함수를 사용하기

```
SELECT department, MIN(name), AVG(salary)
FROM employees
GROUP BY department;
```

- `name` 대신 `MIN(name)`을 사용!
- `MIN(name)`은 알파벳 순으로 가장 앞선 이름을 가져옵니다. → 이렇게 **규칙**을 정해주면 SQL이 헷갈리지 않아요.

3. HAVING vs WHERE

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

- `HAVING` 절은 `GROUP BY` 에 의한 결과를 필터링 할 때 사용됩니다.
- SQL 구문에서는 `GROUP BY` 절 뒤에 위치해요.
- 데이터 필터링이라는 기능을 가지고 있는 `WHERE` 절과의 차이점은 아래와 같아요.
  - **`WHERE` 절은 `GROUP BY` 전 데이터를 기준으로 필터링**
  - **`HAVING` 절은 `GROUP BY` 후 결과값을 기준으로 데이터를 필터링합니다.**

필터링 구문	필터링 시점	설명
`WHERE`	`GROUP BY` **전**	원본 데이터에서 조건을 걸어 **미리 행을 필터링**
`HAVING`	`GROUP BY` **후**	집계가 끝난 **그룹 결과에 조건**을 걸어 필터링

```
SELECT
  기준컬럼,                    -- 그룹 기준
  집계함수1(조건컬럼) AS 별칭1,
  집계함수2(조건컬럼) AS 별칭2
FROM 테이블명
WHERE 조건                     -- (선택 사항)
GROUP BY 기준컬럼              -- 그룹 기준은 SELECT와 동일해야 함
HAVING 집계함수를 이용한 조건식   -- 그룹화된 결과에 대한 조건
;
```

▼ **`WHERE` / `SELECT` / `HAVING` 절의 사용 기준 정리**

1. **`WHERE` 절**
   - **목적**: 데이터베이스에서 **원본 데이터(행)를 걸러내는 조건**을 지정할 때 사용함.
   - **적용 시점**: `SELECT` 실행 이전, 즉 **집계(aggregate) 함수 실행 전**에 필터링
   - **예시1)** 20세 이상인 학생만 선택

     ```
     SELECT name, age
     FROM students
     WHERE age >= 20;
     ```

   - **예시2)** 급여(`salary`)가 5000 이상인 직원만 대상으로 부서별 평균을 구하기

     ```
     SELECT department_id, AVG(salary) AS avg_sal
     FROM employees
     WHERE salary >= 5000     -- 그룹화 전 필터링
     GROUP BY department_id;
     ```

     1. `salary >= 5000` 인 행만 남김
     2. 그 행들만 `department_id` 기준으로 그룹핑

3. 그룹별 평균 계산

2. `SELECT` **절**

- **목적**: **출력할 컬럼(열)을 선택**하거나, 계산/가공된 값을 보여줄 때 사용함.
- **적용 시점**: WHERE로 행이 필터링된 후, **최종적으로 결과에 보여질 열을 결정**.

```
SELECT name, AVG(score) AS avg_score
FROM students
GROUP BY name;
```

→ 이름과 평균 점수만 결과에 표시.

3. `HAVING` **절**

- **목적**: `GROUP BY` **로 집계된 결과에 조건을 걸 때** 사용.
- **적용 시점**: 집계(Aggregation) 이후에 실행 → `WHERE`와 달리 집계 함수(AVG, SUM, COUNT 등)를 조건식에서 사용 가능

  ▼ 참고

  - `WHERE` **절**
    - 그룹화 이전에 행 단위로 조건을 거는 단계
    - 따라서 집계함수(AVG, SUM, COUNT ...) 는 아직 계산되지 않았으므로 사용 불가
    - `WHERE` 조건식에는 컬럼 값, 상수, 서브쿼리만 올 수 있음
  - `HAVING` **절**
    - `GROUP BY` **이후** 집계 결과가 나온 뒤에 조건을 거는 단계 → 집계함수 사용 가능
    - 즉, 그룹 단위 조건 (`AVG(score) >= 80`) 같은 게 가능
- **예시1)** 반별 평균 점수가 80 이상인 반만 출력함.

```
SELECT class, AVG(score) AS avg_score
FROM students
GROUP BY class
HAVING AVG(score) >= 80;
```

- **예시2)** 부서별 평균 급여를 구하되, 그 평균이 5000 이상인 부서만 출력하기

```
SELECT department_id, AVG(salary) AS avg_salary
FROM employees
GROUP BY department_id
HAVING AVG(salary) >= 5000;
```

1. 전체 직원 데이터를 `department_id` 기준으로 그룹핑
2. 각 그룹의 `AVG(salary)` 계산
3. 평균이 5000 이상인 그룹만 남김

4. 서브쿼리 개념과 사용 방법

👋

- 쿼리속에 쿼리! 쿼리를 구조화 해봅시다!
- 참고자료: https://persistent-polyanthus-3e0.notion.site/SQL-3-24722d577d0e80f8b0f4d8060509a147

4-1. WHERE 절 서브쿼리 = 중첩(일반) 서브쿼리

- 서브쿼리 결과에 따라 바깥쪽 `WHERE` 조건이 결정되는 방식
  1. 서브쿼리가 먼저 실행되어 **값을 반환하고,**
  2. 메인쿼리는 **그 반환된 값**을 기준으로 `WHERE` 조건을 적용한다는 뜻!

항목	설명
위치	`WHERE` 절에서 사용됨
목적	서브쿼리 결과에 따라 `WHERE` 조건을 결정 (서브쿼리의 결과에 따라 달라지는 `WHERE` 조건절)
형태	`WHERE 컬럼 = (SELECT ...)`

- `theglory` 테이블

날짜	이름	성별	나이	직업
2024-01-01	박연진	F	30	아나운서
2024-01-02	문동은	F	40	선생님
2024-01-03	전재준	M	25	사장
2024-01-03	최혜정	F	33	승무원
2024-01-04	이사라	F	50	화가
NULL	주여정	M	41	의사
NULL	손명오	M	45	배달원

```
-- 직접 조건을 정함

SELECT 이름
FROM basic.theglory
WHERE 나이 = 50
;
```

- `나이 = 50`이라고 **고정된 조건**이죠.

```
-- 서브쿼리 활용
-- 가장 나이가 많은 사람의 이름 찾기

SELECT 이름
FROM basic.theglory
WHERE 나이 = (
  SELECT MAX(나이)
  FROM basic.theglory
);
```

- 여기서 `MAX(나이)`의 값이 얼마인지 모르지만, 서브쿼리 결과에 따라 `WHERE` 조건이 정해져요.
- 예를 들어 서브쿼리가 실행됐을 때 `MAX(나이)`가 `50`이면 `WHERE` 조건이 `나이 = 50`이 되는 거고, 만약 `55`면 `나이 = 55`가 되는 거예요.

WHERE 절 서브쿼리 = 중첩(일반) 서브쿼리 기본 작성법

- 기본 작성법은 익숙해질 때까지 계속 봐주셔야 합니다! 🙏

```
SELECT 컬럼1, 컬럼2 ...
FROM 테이블명
WHERE 컬럼명 비교연산자 (
```

```
    SELECT 컬럼명
    FROM 테이블명
    WHERE 조건
);
```

WHERE 절 서브쿼리 = 중첩(일반) 서브쿼리는 보통 어떻게 활용하나요?

1. **활용 목적**

- 특정 조건을 만족하는 데이터만 가져오고 싶을 때 사용함.
- **행을 걸러낼 때 기준값을 동적으로 뽑아와야 하는 경우**
- 이 조건에 해당하는 값이 다른 테이블에도 있는가? 같은 상황에 유용함.

2. **키워드**

- **보다 크다/작다 (`>`, `<`, `=`, `!=`)**
  - 단순 비교인데, 비교할 대상이 **고정값이 아니라 쿼리 결과**일 때 → 서브쿼리 필요
  - 예시) 전체 평균보다 큰 급여

```
WHERE salary > (SELECT AVG(salary) FROM employees)
```

- **소속 여부 (`IN`, `NOT IN`)**
  - ~ 목록에 속하는지 확인 → 그 **목록을 만들어내는 쿼리**가 서브쿼리
  - 예시) 부서가 '영업부'에 속한 직원만

```
WHERE department_id IN (SELECT department_id FROM departments WHERE name='영업부')
```

- **존재 여부 (`EXISTS`, `NOT EXISTS`)**
  - 관련 데이터가 존재하면/존재하지 않으면 → **서브쿼리의 결과 존재 유무 체크**
  - 예시) 프로젝트를 배정받은 직원만

```
WHERE EXISTS (SELECT 1 FROM project_assignment p WHERE p.emp_id = e.emp_id)
```

- **최대/최소와 비교 (`MAX`, `MIN`)**
  - 최대값/최소값을 가진 행만 찾아라 → **그 최대·최소값을 뽑는 서브쿼리 필요**
  - 예시) 급여가 최고인 직원

```
WHERE salary = (SELECT MAX(salary) FROM employees)
```

- **평균보다 큰/작은 (`AVG`)**
  - 평균을 기준으로 비교 → 평균값을 내는 **집계 서브쿼리** 필요
  - 예시) 평균보다 급여가 높은 직원

```
WHERE salary > (SELECT AVG(salary) FROM employees)
```

👋
- **비교 기준값**이 단일 상수값이면 그냥 `WHERE salary > 5000` 같은 단순 조건
- 그런데 그 기준값이 **동적으로 계산된 값**(최대/최소/평균/목록/존재 여부)이면 → **WHERE 절 서브쿼리**를 떠올려라

3. **예시**

- 학생 테이블(`STUDENT`)과 장학금 수혜자 테이블(`SCHOLARSHIP`)이 있다고 할 때, 장학금을 받은 학생만 조회하고 싶다면?

```
SELECT NAME
FROM STUDENT
WHERE ID IN (SELECT STUDENT_ID FROM SCHOLARSHIP);
```

▼ 쿼리 해석

1. 안쪽 서브쿼리 먼저 실행

```
SELECT STUDENT_ID FROM SCHOLARSHIP;
```

- `SCHOLARSHIP` 테이블에서 **장학금을 받은 학생들의 ID**만 뽑습니다.
- 결과: `[1001, 1003, 1005, ...]` 처럼 ID 목록(집합)이 나옴.

2. 바깥 쿼리 실행

```
SELECT NAME
FROM STUDENT
WHERE ID IN (... 안쪽 결과 ...);
```

- `STUDENT` 테이블에서 학생 이름(`NAME`)을 가져오되,
- **그 학생의 ID가 안쪽 서브쿼리에서 나온 ID 집합 안에 있는 경우만** 선택합니다.

✅ **포인트:** `WHERE` 서브쿼리는 위와 같이 **필터링 조건**으로 활용하는 경우가 대다수

조건이 있을때 서브쿼리가 WHERE 절이나 HAVING 절에서 어떻게 쓰이는지?

1. WHERE 절에서 서브쿼리

- `WHERE` 절은 **행(Row) 단위**로 조건을 거는 곳이죠? 즉, 집계가 일어나기 전에 필터링을 합니다.
- 예시) **평균 급여보다 높은 사람만 조회**

```
SELECT name, salary
FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);
```

- `(SELECT AVG(salary) FROM employees)` → 전체 직원의 평균 급여를 하나의 값으로 계산
- 그 값을 `WHERE salary > ...` 조건에 넣어서, 평균보다 높은 직원만 남김
- `WHERE`에서 서브쿼리는 **행 하나 하나와 비교할 기준값**을 만들어줄 때 유용하게 사용합니다.

2. HAVING에서 서브쿼리

- `HAVING` 절은 `GROUP BY` **로 묶은 그룹 단위**에 조건을 거는 곳인거 아시죠? 즉, 집계가 끝난 후에 필터링합니다.
- 예시) 전체 평균 급여보다 부서 평균이 높은 부서만 조회

```
SELECT department, AVG(salary) AS avg_salary
FROM employees
GROUP BY department
HAVING AVG(salary) > (SELECT AVG(salary) FROM employees);
```

- 먼저 `GROUP BY department` 로 부서별 평균 급여 계산
- `HAVING` 에서 서브쿼리를 이용해 전체 평균과 비교 → 평균보다 높은 부서만 결과에 남음.
- `HAVING` 에서 서브쿼리를 쓰면, "그룹별 집계 결과"를 "전체 기준값"과 비교할 수 있음.
  - avg_salary: 그룹별 집계 결과
  - SELECT AVG(salary) FROM employees: 전체 기준값
- `HAVING` 에서 서브쿼리는 **그룹 단위 조건**을 줄 때 사용합니다.

3. 정리

구분	WHERE절 서브쿼리	HAVING절 서브쿼리
조건 대상	**행(Row)**	**그룹(Group)**
실행 시점	집계 **이전**	집계 **이후**
서브쿼리 역할	각 행과 비교할 값 제공	그룹 집계 결과와 비교할 값 제공

4-2. FROM 절 서브쿼리 = 인라인 뷰(가장 많이 사용) ⭐

- 인라인 뷰는 가장 자주 사용되는 서브쿼리 유형 중 하나이며, **특히 복잡한 조인이나 집계, 필터링을 깔끔하게 구성할 때 매우 유용합니다.**

항목	설명
정의	`FROM` 절에 사용되는 서브쿼리로, 하나의 테이블처럼 사용
위치	`FROM` 절 내부
형태	`(SELECT ...) AS 별칭`
별칭(AS)	**반드시 지정해야 함. (** `AS 별칭` 없으면 외부에서 참조 불가)
주요 사용처	`JOIN` , `UNION` 시 가장 유용하게 사용됨.

인라인 뷰 서브쿼리 기본 작성법

- 기본 작성법은 익숙해질 때까지 계속 봐주셔야 합니다! 🙏

```
SELECT 별칭1.컬럼1, 별칭1.컬럼2, 별칭2.컬럼3
FROM (
    SELECT 컬럼1, 컬럼2
    FROM 테이블명
    WHERE 조건
) AS 별칭1
JOIN (
    SELECT 컬럼3, 컬럼4
    FROM 테이블명
    WHERE 조건
) AS 별칭2
ON 별칭1.공통컬럼 = 별칭2.공통컬럼;
```

예제) 나이가 33세 이상인 모든 데이터 중 나이와 직업 컬럼 반환하기

- `theglory` 테이블

날짜	이름	성별	나이	직업
2024-01-01	박연진	F	30	아나운서
2024-01-02	문동은	F	40	선생님
2024-01-03	전재준	M	25	사장
2024-01-03	최혜정	F	33	승무원
2024-01-04	이사라	F	50	화가
	주여정	M	41	의사
	손명오	M	45	배달원

```
SELECT
   x.나이,
   x.직업
FROM (
   SELECT *
   FROM basic.theglory
   WHERE 나이 >= 33
) AS x;
```

- 쿼리 해석
  - `x`는 서브쿼리(인라인 뷰)의 별칭(alias)이고, 괄호 안에 있는 서브쿼리 전체를 대표하는 이름이에요.
  - 즉, 그 서브쿼리의 결과를 `x`라는 이름으로 부르겠다는 의미입니다.
  - 그 이후에는 `x.컬럼명` 형식으로 마치 테이블처럼 다룰 수 있게 되는 거죠. (하나의 테이블처럼 사용)
- 쿼리 실행 결과

SELECT x.나이, x.직업 FROM (S Er

	123 나이	A-Z 직업
1	40	선생님
2	33	승무원
3	50	화가
4	41	의사
5	45	배달원

FROM 절 서브쿼리 = 인라인 뷰 서브쿼리는 보통 어떻게 활용하나요?

1. 집계 결과를 재활용

- 대표적인 활용은 **중간 집계**예요.
- 한 번 `GROUP BY`로 묶어서 평균, 합계, 순위 같은 값을 구한 후 그 결과를 가상 테이블처럼 활용합니다.
- **예시) 부서별 평균 급여보다 많이 받는 직원 찾기**

```
SELECT E.EMP_NAME, E.SALARY, T.AVG_SAL
FROM (
   SELECT DEPT_ID, AVG(SALARY) AS AVG_SAL
   FROM EMP
   GROUP BY DEPT_ID
) T
```

```
JOIN EMP E ON E.DEPT_ID = T.DEPT_ID
WHERE E.SALARY > T.AVG_SAL;
```

✅ **포인트**: `T`라는 인라인 뷰를 만들어 두고, 바깥 쿼리에서 활용.

2. **복잡한 계산 단순화**

- 쿼리를 나눠서 읽기 쉽게 만들 수 있습니다.
- 긴 쿼리를 한 번에 쓰면 가독성이 떨어지는데, 인라인 뷰로 쪼개면 중간 결과 테이블을 보는 것처럼 단순해져요.
- **예시) 최근 3개월치 매출만 추려서 평균 계산**

```
SELECT AVG(SALES) AS AVG_SALES
FROM (
   SELECT *
   FROM SALES
   WHERE SALE_DATE >= ADD_MONTHS(SYSDATE, -3)
) LAST3
;
```

✅ **포인트**: 서브쿼리에서 최근 3개월만 걸러놓고, 바깥 쿼리에서는 평균만 구함.

3. **순위/Top-N 문제 해결**

- RANK, ROW_NUMBER 같은 윈도우 함수와 함께 많이 쓰입니다.
- "상위 3명", "과목별 1등" 같은 문제를 풀 때 인라인 뷰가 유용해요.
- **예시) 과목별 1등 학생 뽑기**

```
SELECT *
FROM (
   SELECT STUDENT_ID, SUBJECT, SCORE,
        ROW_NUMBER() OVER (PARTITION BY SUBJECT ORDER BY SCORE DESC) AS RN
   FROM GRADE
) T
WHERE RN = 1;
```

✅ **포인트**: 인라인 뷰에서 순위를 매겨 놓고, 바깥 쿼리에서 1등만 뽑음.

4. **여러 테이블/쿼리 합치기**

- UNION, 복잡한 JOIN 결과를 **인라인 뷰로 감싸서 또 다른 JOIN이나 필터링**에 활용.
- 즉, **서브쿼리 결과를 하나의 테이블처럼 사용!**

1. 상황 가정

- `SALES_2024`, `SALES_2025` 두 개의 테이블에 연도별 매출 데이터가 따로 저장돼 있음.
- 우리는 두 테이블을 **UNION으로 합쳐서 하나의 가상 테이블**을 만든 뒤, 그 결과에서 **상품별 총 매출**을 구하고 싶음.

2. 예시 쿼리1

- **UNION 합친 결과 → 인라인 뷰로 감싸서 집계/필터링**

```
SELECT PRODUCT_ID, SUM(AMOUNT) AS TOTAL_SALES
FROM (
   SELECT PRODUCT_ID, AMOUNT
   FROM SALES_2024
   UNION ALL
   SELECT PRODUCT_ID, AMOUNT
   FROM SALES_2025
) AS ALL_SALES
GROUP BY PRODUCT_ID;
```

▼ 예시 쿼리1 설명

1. `UNION ALL` **부분**
   - `SALES_2024`와 `SALES_2025` 데이터를 합쳐서 **ALL_SALES**라는 가상의 테이블을 만듦.
   - 이게 바로 **FROM 절 인라인 뷰**
2. **바깥 쿼리**
   - 이제는 `ALL_SALES`를 **하나의 통합 테이블처럼** 다룰 수 있음.
   - 그 위에서 `GROUP BY`를 써서 `PRODUCT_ID`별 총 매출을 계산.

✅ **포인트: 여러 개 쿼리의 결과를 합친 뒤, 그걸 또 다른 쿼리에서 테이블처럼 쓰는 것**이에요.

3. 예시 쿼리2 (JOIN 결과 감싸기)

- **JOIN 결과 → 인라인 뷰로 감싸서 또 다른 집계/연산**

```
SELECT DEPT_ID, AVG(SALARY) AS AVG_SAL
FROM (
   SELECT E.EMP_ID, E.DEPT_ID, D.DEPT_NAME, E.SALARY
   FROM EMP E
   JOIN DEPT D ON E.DEPT_ID = D.DEPT_ID
) AS EMP_DEPT
GROUP BY DEPT_ID;
```

- `EMP`와 `DEPT`를 JOIN해서 만든 결과를 **EMP_DEPT 인라인 뷰**로 만들고, 그걸 다시 이용해서 부서별 평균 급여를 구함.

✅ **정리**

구분	UNION 활용 (여러 테이블 합치기)	JOIN 활용 (여러 테이블 결합)
상황	같은 구조의 데이터를 여러 테이블에서 가져와 합쳐야 할 때	두 테이블을 조인해서 확장된 데이터를 만들고, 그 결과를 다시 활용할 때
용도	연도별 / 지역별 / 기간별 분리된 테이블을 합쳐서 하나의 데이터셋으로 재활용	조인 결과를 다시 집계, 필터링, 정렬, 순위 계산 등에 활용
포인트	`UNION`은 **비슷한 구조(같은 컬럼 구조)를 가진 여러 테이블**을 "하나로 합쳐서" 마치 **통합 테이블처럼** 만들어 줌.	`JOIN`은 서로 **다른 정보를 가진 테이블**을 연결해서 "정보가 확장된 하나의 테이블"을 만들어 줌.

구분	UNION 활용 (여러 테이블 합치기)	JOIN 활용 (여러 테이블 결합)
	- 예시) 2024년 매출 테이블 + 2025년 매출 테이블 → "전체 매출 테이블"	- 예시) 직원 테이블 + 부서 테이블 → "직원+부서 정보 테이블"
비유	**두 반의 성적표를 합쳐서 전체 학급 성적표**를 만드는 것	**학생 명단 + 주소록을 합쳐서, '누가 어디 사는지'까지 알 수 있는 명단**을 만드는 것

- **FROM 절 서브쿼리는 가상 테이블처럼 동작!!**
- 쿼리 안에서 만든 이 임시 결과를 **바로 이어서 다른 연산에 활용**할 수 있다.
  - 조건절에서 사용하고,
  - 다른 테이블과 조인하고,
  - 더 복잡한 계산을 이어갈 수 있다.

☝️

단, 이 가상 테이블은 **해당 쿼리 내에서만 유효하며 재사용은 불가능**하다. (재사용은 `WITH` 구문에서 가능)

4-3. SELECT절 서브쿼리 = 스칼라 서브쿼리

- **스칼라(scala) 값 = 단일 값**

항목	설명
사용 위치	`SELECT` 절에서 사용
특징	서브쿼리 결과가 단 하나의 값(1개의 행, 1개의 열)을 반환해야 함.
사용 방식	**하나의 컬럼처럼** 다른 컬럼들과 함께 사용
주의사항	일반적으로 **다른 테이블과 함께 사용할 때** 의미가 있음. - 스칼라 서브쿼리 이용을 위해서는, 서로 다른 테이블이 필요함.

SELECT절 서브쿼리 = 스칼라 서브쿼리 기본 작성법

- 기본 작성법은 익숙해질 때까지 계속 봐주셔야 합니다! 🙏

```
SELECT
   컬럼1,
   컬럼2,
   (
       SELECT 컬럼명
      FROM 테이블명
      WHERE 조건
   ) AS 별칭
FROM 테이블명;
```

예제) `theglory` 테이블(별칭 a)에서 `이름`, `나이`를 가져오고 `theglory2` 테이블(별칭 b)에서 `이름`이 같은 사람의 출현 횟수 (`COUNT`), 결제 총액 (`SUM`)를 구해봅시다.

- `theglory` 테이블의 이름과 `theglory2` 테이블의 이름이 일치하는 경우를 `count` 하여, `same_name_cnt` 컬럼으로 반환
- `theglory` 의 이름과 `theglory` 테이블의 이름이 일치하는 경우의 결제금액을 `sum` 하여, `same_name_sumamount` 컬럼으로 반환

[`theglory` 테이블]

날짜	이름	성별	나이
2024-01-01	박연진	F	30
2024-01-02	문동은	F	40

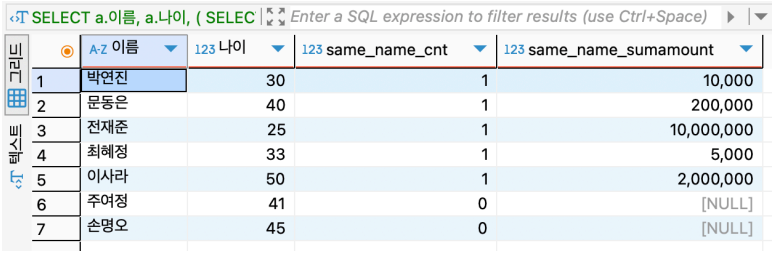
[`theglory2` 테이블]

이름	결제금액
박연진나운서	10,000
문동은생님	200,000

날짜	이름	성별	나이	이름	결제금액
2024-01-03	전재준	M	25	전재준	10,000,000
2024-01-03	최혜정	F	33	최혜정	5,000
2024-01-04	이사라	F	50	이사라	2,000,000
	주여정	M	41		의사
	손명오	M	45		배달원

```
SELECT
  a.이름,
  a.나이,
  (
    SELECT COUNT(*)
    FROM basic.theglory2 b
    WHERE b.이름 = a.이름
  ) AS same_name_cnt,
  (
    SELECT SUM(b.결제금액)
    FROM basic.theglory2 b
    WHERE b.이름 = a.이름
  ) AS same_name_sumamount
FROM basic.theglory a;
```

- 쿼리 실행 결과



"하나의 컬럼처럼 쓰인다"는 말은 스칼라 서브쿼리가 딱 하나의 값을 반환하기 때문에, 메인쿼리의 SELECT 절의 컬럼(열)에 들어갈 수 있다는 뜻이에요.

스칼라 쿼리는 한 개의 값을 반환하며, 그 값을 기준 테이블의 각 행에 반복적으로 붙여주는 쿼리입니다.

- **해당 쿼리는 "전체 결과가 1행만 나온다"가 아니라, `basic.theglory a` 의 행 수만큼 결과가 나옵니다.**
- 왜냐하면 바깥 SELECT의 **출력 행 개수는 테이블 `a` 의 행 수**로 결정되고, 괄호 안의 서브쿼리들은 각 행마다 계산되는 "스칼라 값(단일 값)"을 만들어 붙이기 때문이에요.

SELECT 절 서브쿼리 = 스칼라 서브쿼리는 보통 어떻게 활용하나요?

1. **핵심 아이디어**

- **스칼라 서브쿼리** = 각 행마다 하나의 값을 붙이거나 비교할 때 사용
- **SELECT 절 → 최종적으로 출력할 컬럼 / 표현식을 선택**
  - SELECT 절에 두면, 열을 하나 더 만드는 것처럼 쓸 수 있음.

2. 예시1) 학생 이름과 학과 이름 같이 보여주기

- **테이블**
  - `STUDENT(STUDENT_ID, NAME, DEPT_ID)`
  - `DEPARTMENT(DEPT_ID, DEPT_NAME)`

```
SELECT S.NAME,
       (SELECT D.DEPT_NAME
        FROM DEPARTMENT D
        WHERE D.DEPT_ID = S.DEPT_ID) AS DEPARTMENT_NAME
FROM STUDENT S;
```

- 쿼리 실행 결과

NAME	DEPARTMENT_NAME
철수	컴퓨터공학과
영희	경영학과
민수	기계공학과

✅ **포인트**: 학생 테이블만 조회했는데, 스칼라 서브쿼리 덕분에 "학과 이름"을 붙일 수 있음.

3. 예시 2) 직원과 '부서 평균 급여' 같이 보여주기

- **테이블**
  - `EMP(EMP_ID, EMP_NAME, SALARY, DEPT_ID)`

```
SELECT E.EMP_NAME, E.SALARY,
       (SELECT AVG(SALARY)
        FROM EMP
        WHERE DEPT_ID = E.DEPT_ID) AS DEPT_AVG_SAL
FROM EMP E;
```

- 쿼리 해석
  1. 바깥쪽 `FROM EMP E` 에서 **직원 한 행을 집어 듭니다.** (예: 홍길동, 급여 300, 부서 10).
  2. 그 행의 `DEPT_ID` 를 들고 안쪽 서브쿼리를 실행합니다.
     - `SELECT AVG(SALARY) FROM EMP WHERE DEPT_ID = E.DEPT_ID`
     - 즉, 부서 10의 모든 직원 급여 평균을 계산합니다.
  3. 서브쿼리는 항상 한 값(스칼라 값)을 반환 → 그 값을 `DEPT_AVG_SAL` 열로 붙입니다.
  4. 이 과정을 EMP 테이블의 **모든 행에 대해 반복**합니다.

- 쿼리 실행 결과

EMP_NAME	SALARY	DEPT_AVG_SAL
홍길동	300	350
김철수	400	350
박영희	350	350

✅ **포인트**: 각 행마다 자기 부서 평균(DEPT_AVG_SAL)이 계산돼서 열처럼 붙는다 = SELECT 절 스칼라 서브쿼리의 전형적인 쓰임. (각 직원의 급여와 **자기 부서 평균 급여**를 나란히 보여줄 수 있게 됨.)

- 같이 보여라 / 옆에 붙여라 ← 보통 이런 느낌?!
- 각 행마다 비교용 수치 필요
- 전체 평균/전체 합계도 출력
- 다른 테이블에서 가져온 값 컬럼 추가

- 각 행과 전체 집계 결과 비교
- 추가 통계값, 순위, 집계 수를 같이 출력
- 다른 테이블의 특정 값 붙이기