M05-02 · GIBALJA VISUAL MANUAL

# SQL로 데이터를 조회·변경하기

> **한 문장 목표:** SQL을 문법 암기가 아니라 `질문 → source → relation → row → group → output → order/change → 증거` 의 흐름으로 읽고, 조회는 예상 결과로, 변경은 되돌릴 수 있는 증거로 마무리합니다.

| 난이도 | 개념 | 실습 | 셀프 테스트 | 최종 산출물 |
|---|---|---|---|---|
| Level 1 | 60분 | 60분 | 15분 | SQL 실습 기록, 결과 검증표, JOIN·집계 설명, 변경·rollback 체크리스트 |

"SQL을 실행했는데 오류가 없었다"는 정답의 증거가 아닙니다. 조회는 틀린 row를 그럴듯하게 반환할 수 있고, 변경은 WHERE 하나가 빠져 전체 row를 바꿀 수 있습니다. 실행 전에 예상 row·group 수를 말하고, 실행 후에 column·count·sample·RETURNING·transaction 결과를 남기는 습관이 데이터를 지킵니다.

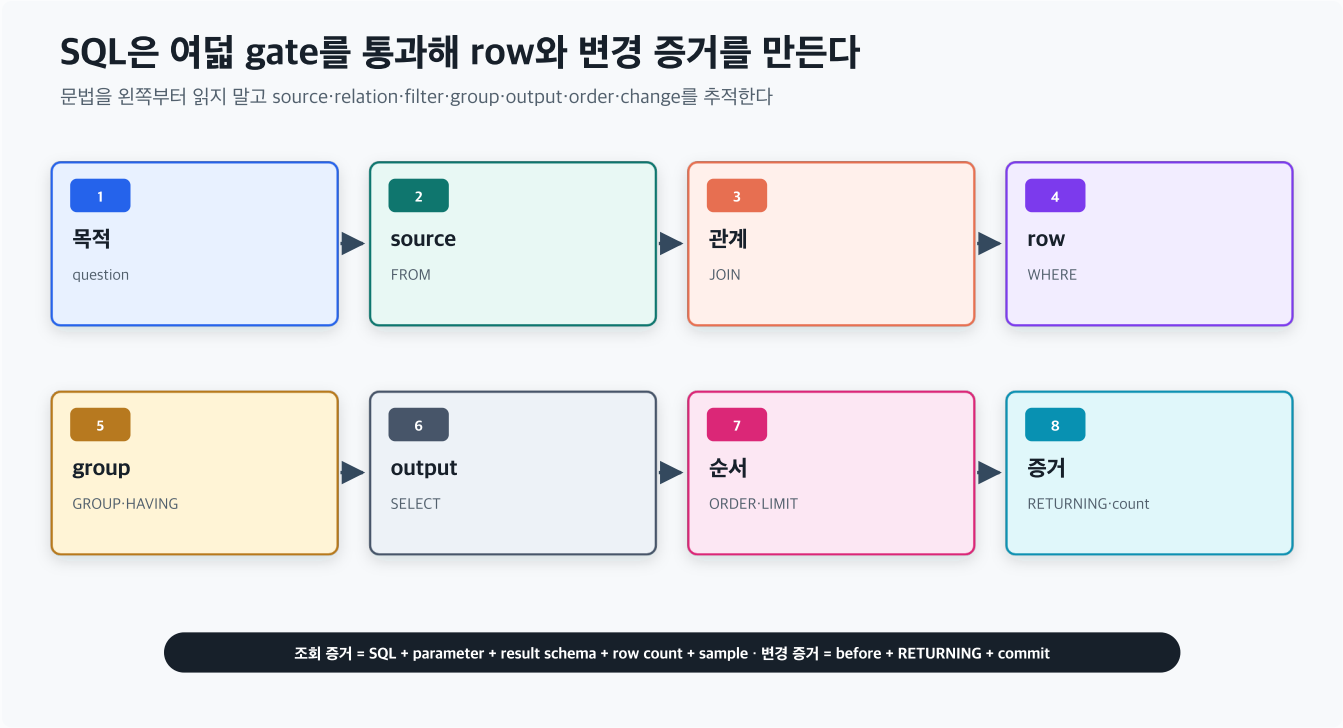
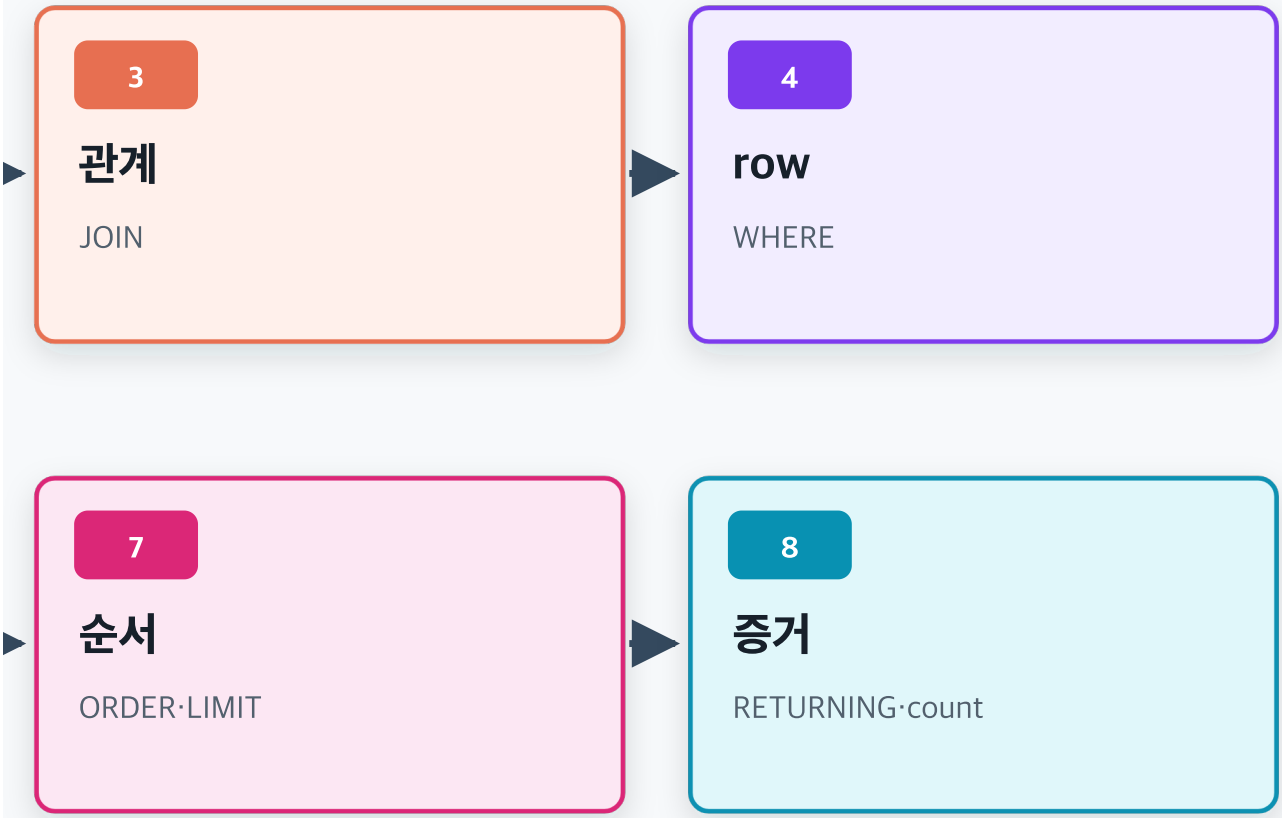



그림 1. SQL은 문장 하나가 아니라 여덟 판단의 연결입니다. 앞 gate의 row 범위가 뒤 gate의 집계·순서·변경 범위를 결정합니다.


1
목적
question
2
source
FROM
5
group
GROUP·HAVING
6
output
SELECT
조회 증거 = SQL + parameter + result schema + row cou




nt + sample · 변경 증거 = before + RETURNING + commit

# 0. 이 PDF를 공부하는 방법

## 1회차 · 그림만 읽기 · 20분

모든 그림의 제목과 맨 아래 결론 띄만 읽습니다. 다음 여덟 문장을 소리 내어 말합니다.

```
결과의 한 row가 무엇을 뜻하는지 먼저 정한다.
WHERE는 TRUE인 row만 남기고 NULL 비교의 UNKNOWN은 버린다.
JOIN 후 row 수는 관계의 1·N과 ON·WHERE에 따라 바뀐다.
WHERE는 input row, HAVING은 aggregate group을 거른다.
LIMIT은 순서를 만들지 않고 ORDER BY가 순서를 정한다.
값은 parameter로 보내고 SQL 문자열에 연결하지 않는다.
UPDATE·DELETE는 같은 WHERE의 SELECT로 대상을 먼저 확인한다.
변경은 RETURNING·row count·COMMIT 또는 ROLLBACK을 증거로 남긴다.
```

## 2회차 · row를 손으로 추적하기 · 25분

그림 3의 학습 dataset을 종이에 적고 다음 질문에 실행 전 답합니다.

| 질문 | 실행 전 예상 |
| --- | --- |
| active enrollment는 몇 row입니까 | |
| completion_percent가 NULL인 row는 몇 개입니까 | |
| member와 enrollment를 INNER JOIN하면 몇 row입니까 | |
| course별로 GROUP BY하면 몇 group입니까 | |

## 3회차 · 실제 SQL 엔진에서 실행하기 · 50분

SQL 조회·변경 증거 스튜디오를 열고 20개 시나리오를 순서대로 실행합니다. 각 시나리오에서 SQL을 한 줄만 바꿔 expected·actual이 어떻게 달라지는지 관찰합니다.

## 4회차 · 내 기능의 query·change 증거 작성 · 40분

SQL 조회·변경 증거서를 채웁니다. 최소 조회 2개, 집계 1개, 변경 1개, rollback 1개를 기록합니다.

# 1. SQL은 질문·변경·증거의 계약입니다

SQL 문법이 실행된다는 것과 업무 질문에 맞는다는 것은 다릅니다. 올바른 query는 다음 네 가지를 함께 설명합니다.

| 계약 | 질문 | 증거 |
| --- | --- | --- |
| 결과 grain | 결과의 한 row는 무엇입니까 | `한 row = ...` 문장 |
| 범위 | 어느 source·tenant·기간·상태입니까 | FROM·JOIN·WHERE·parameters |
| 결과 | 어떤 column·row·group·순서입니까 | result schema·count·sample |
| 변경 | 무엇이 몇 row 바뀌었습니까 | target SELECT·RETURNING·transaction outcome |

## SQL을 쓰기 전 한 문장

```
[source]에서 [relation·row 조건]을 만족하는 [grain]을
[column·aggregate]로 보여 주고 [order·limit]로 배치한다.
```

예:

```
취소되지 않은 수강 신청에서 강좌별 신청 수를
많은 순으로 보여 주고 동률은 course_id 순으로 배치한다.
```

## 실행 전 예상 박스

| 항목 | 예상 |
| --- | --- |
| result grain | 한 row = 한 강좌 |
| input row | 취소 제외 신청 3 row |
| group | 2 course groups |
| output column | `course_id`, `learner_count` |
| output row | 2 rows |
| order | `learner_count DESC, course_id ASC` |

**30초 확인 1**

`SELECT * FROM enrollment;`이 오류 없이 실행되었다면, 어떤 업무 질문에 답했다고 말할 수 있습니까?

## 2. 작성 순서와 결과가 만들어지는 논리 순서를 나눴니다

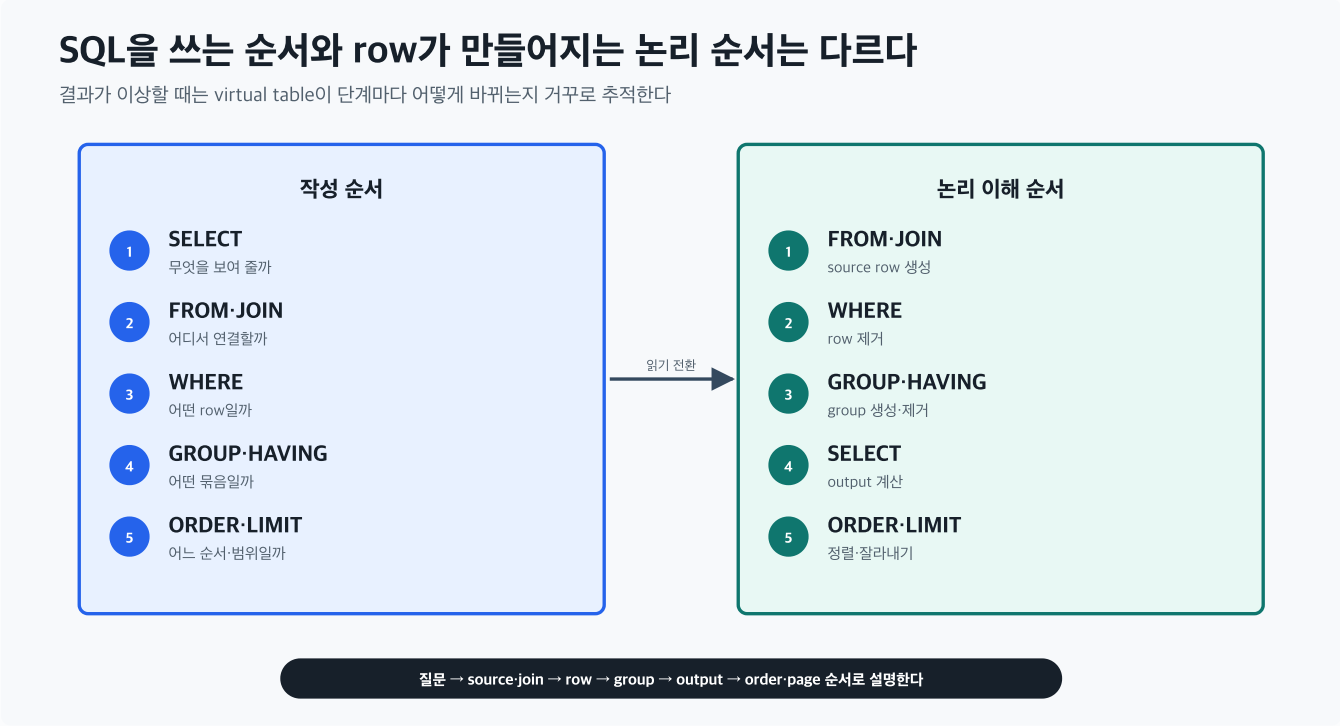
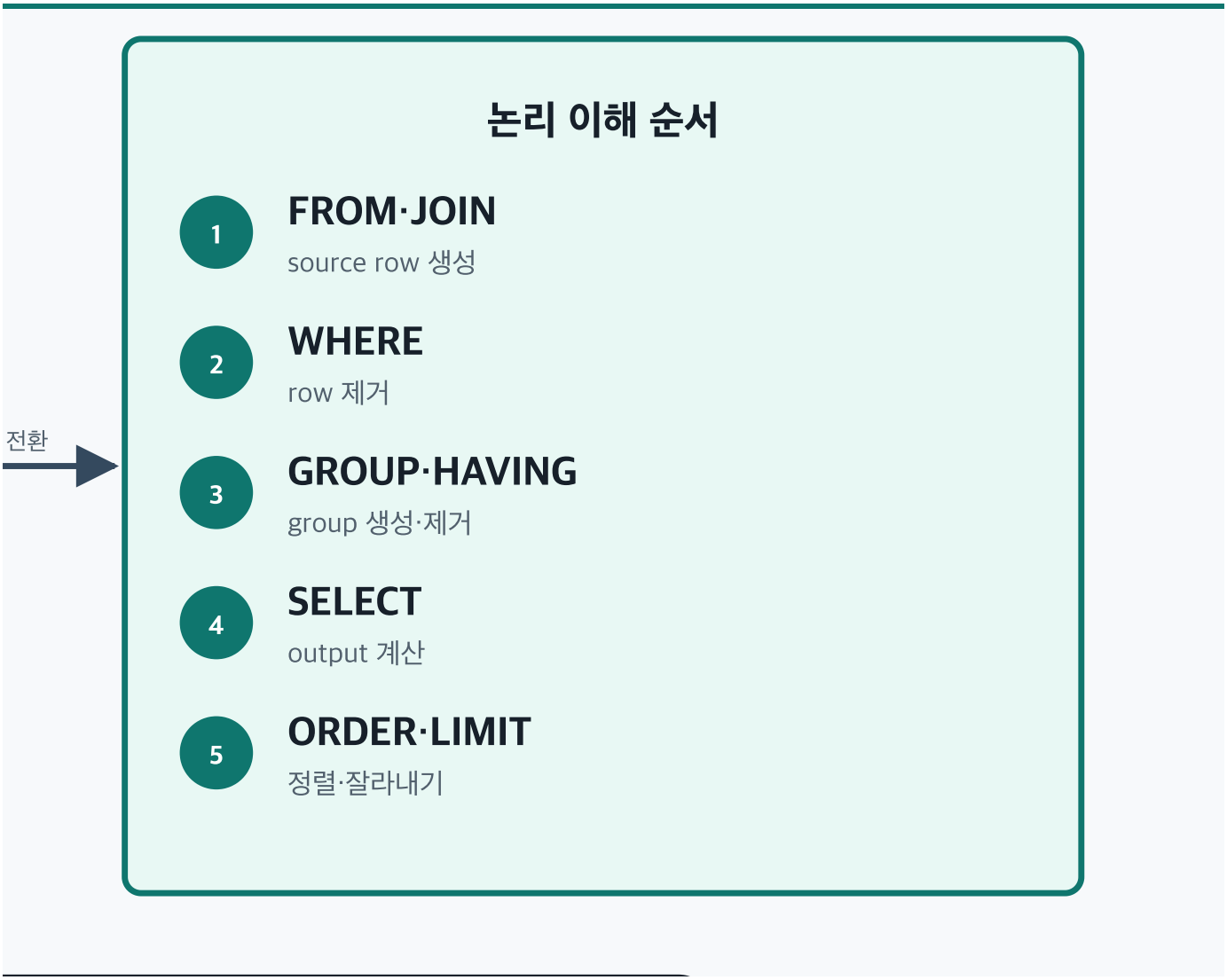



그림 2. SQL은 SELECT부터 쓰지만 결과를 추적할 때는 FROM·JOIN에서 출발합니다. 이 순서는 학습을 위한 논리적 이해 순서이며 DB optimizer의 물리적 실행 순서를 그대로 뜻하지는 않습니다.

## 작성 순서

1. **SELECT**
   무엇을 보여 줄까
2. **FROM·JOIN**
   어디서 연결할까
3. **WHERE**
   어떤 row일까
4. **GROUP·HAVING**
   어떤 묶음일까
5. **ORDER·LIMIT**
   어느 순서·범위일까

읽기


논리 이해 순서
1
FROM·JOIN
source row 생성
2
WHERE
row 제거
전환
3
GROUP·HAVING
group 생성·제거
4
SELECT
output 계산
5
ORDER·LIMIT
정렬·잘라내기

## 2.1. 작성 순서

```
SELECT c.course_id, COUNT(*) AS learner_count
FROM enrollment AS e
JOIN course AS c ON c.course_id = e.course_id
WHERE e.status <> 'canceled'
GROUP BY c.course_id
HAVING COUNT(*) >= 1
ORDER BY learner_count DESC, c.course_id
LIMIT 20;
```

## 2.2. 논리적 이해 순서

1. `FROM·JOIN` : source row를 만듭니다.
2. `WHERE` : 개별 row 중 TRUE만 남깁니다.
3. `GROUP BY` : 남은 row를 group으로 묶습니다.
4. `HAVING` : group 중 조건을 만족하는 group만 남깁니다.
5. `SELECT` : output column·expression을 만듭니다.
6. `ORDER BY·LIMIT` : 결과 순서와 범위를 정합니다.

PostgreSQL 문서는 table expression이 `FROM` 에서 시작해 `WHERE` , `GROUP BY` , `HAVING` 을 거쳐 SELECT list에 전달되는 virtual table을 만든다고 설명합니다. 다만 DB는 같은 결과를 보존하면서 index·join algorithm·predicate pushdown으로 물리 실행 계획을 바꿀 수 있습니다.

## alias가 안 보이는 이유

```
SELECT e.status AS enrollment_status
FROM enrollment AS e
WHERE enrollment_status = 'active'; -- 일반적으로 사용 불가
```

`WHERE` 를 이해하는 시점에는 SELECT list의 alias가 아직 output으로 만들어지지 않았다고 생각하면 이해하기 쉽습니다. 식을 반복하거나 subquery·CTE로 단계를 나눕니다.

# 3. 학습 dataset의 grain·key·NULL을 먼저 봅니다

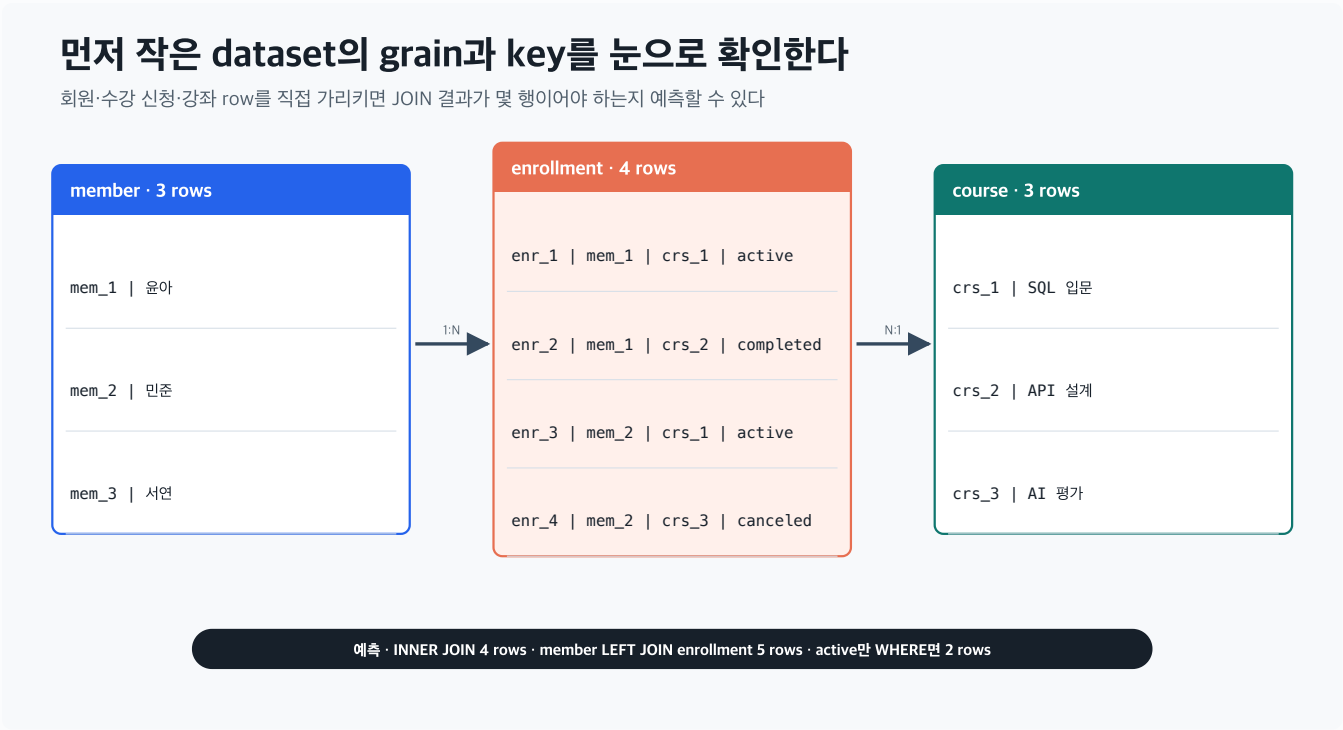
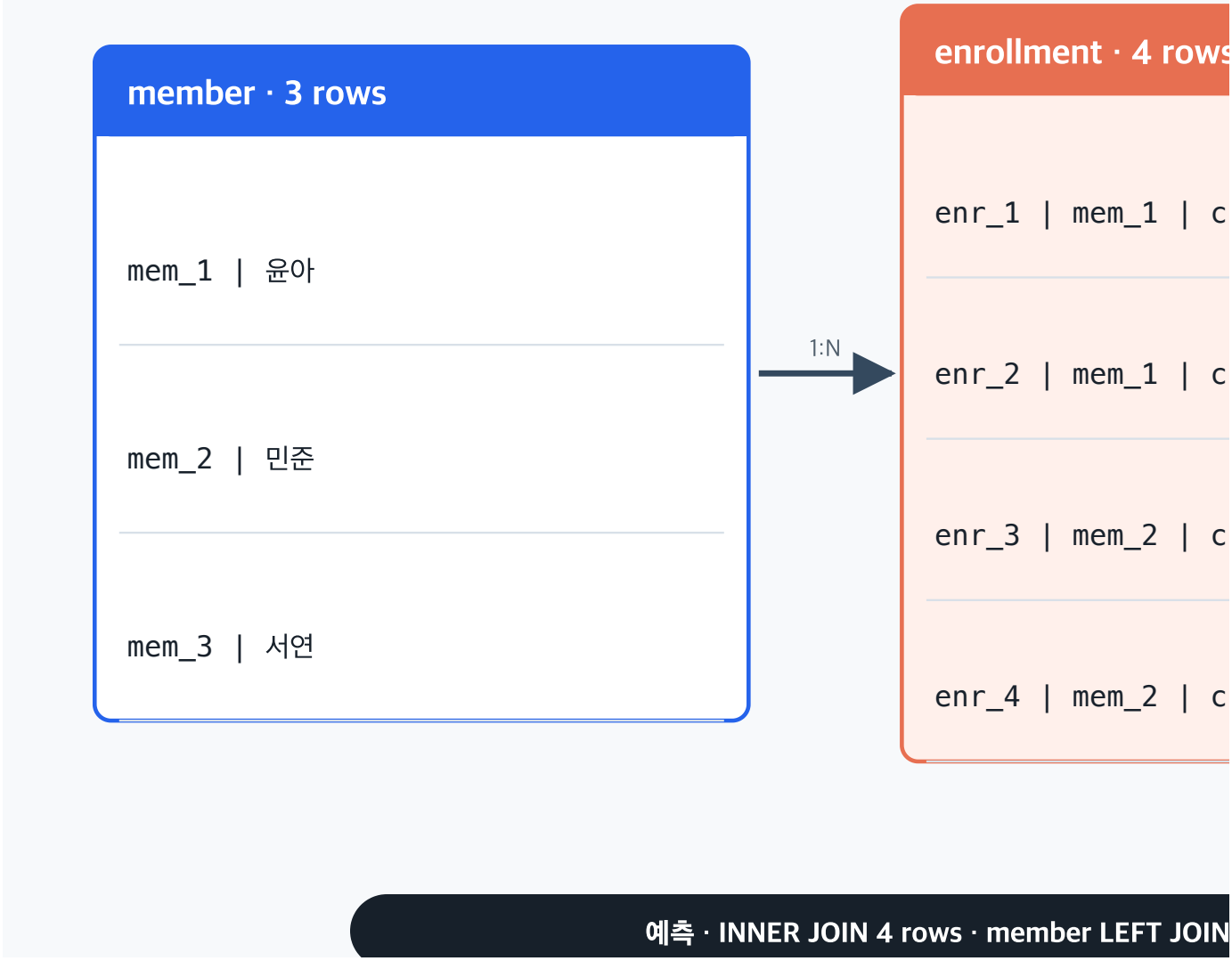
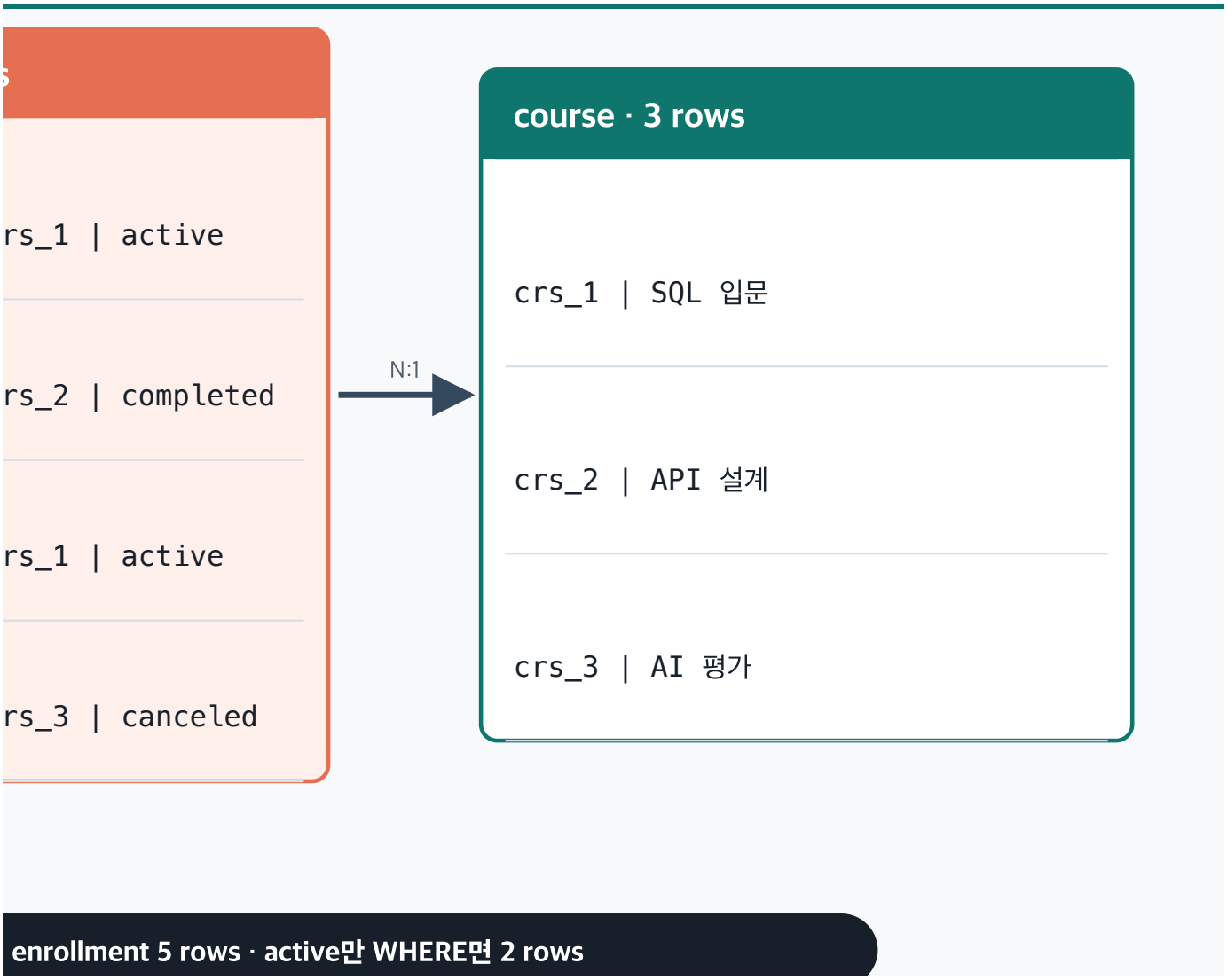



그림 3. 모든 예제는 같은 3개 표에서 출발합니다. SQL을 실행하기 전에 input row 10개를 눈으로 확인합니다.


member · 3 rows
mem_1 | 윤아
mem_2 | 민준
mem_3 | 서연
1:N
enrollment · 4 rows
enr_1 | mem_1 | c
enr_2 | mem_1 | c
enr_3 | mem_2 | c
enr_4 | mem_2 | c
예측 · INNER JOIN 4 rows · member LEFT JOIN


rs_1 | active
rs_2 | completed
rs_1 | active
rs_3 | canceled
N:1
course · 3 rows
crs_1 | SQL 입문
crs_2 | API 설계
crs_3 | AI 평가
enrollment 5 rows · active만 WHERE면 2 rows

## 3.1. member · 한 row = 한 회원

| member_id | name | email | mentor_id |
|---|---|---|---|
| mem_1 | 윤아 | yuna@example.com | NULL |
| mem_2 | 민준 | minjun@example.com | mem_1 |
| mem_3 | 서현 | seohyun@example.com | NULL |

## 3.2. course · 한 row = 한 강좌

| course_id | title | state |
|---|---|---|
| crs_1 | SQL 입문 | active |
| crs_2 | API 설계 | active |
| crs_3 | AI 평가 | draft |

## 3.3. enrollment · 한 row = 한 회원의 한 강좌 신청

| enrollment_id | member_id | course_id | status | completion_percent | created_at |
|---|---|---|---|---|---|
| enr_1 | mem_1 | crs_1 | active | 100 | 2026-07-10 10:00 |
| enr_2 | mem_1 | crs_2 | completed | NULL | 2026-07-10 10:00 |
| enr_3 | mem_2 | crs_1 | active | 80 | 2026-07-11 11:00 |
| enr_4 | mem_2 | crs_3 | canceled | 0 | 2026-07-12 12:00 |

## 실행 전 기준선

```
member rows = 3
course rows = 3
enrollment rows = 4
active enrollment rows = 2
completion_percent IS NULL rows = 1
INNER JOIN enrollment→member→course rows = 4
LEFT JOIN member→active enrollment rows = 3
```

**30초 확인 2**
`member JOIN enrollment`의 결과에서 윤아가 두 번 보이면 중복 data입니까, one-to-many 관계의 정상적 반복입니까?

# 4. SELECT·FROM으로 결과의 한 row와 column을 정합니다

## 4.1. 필요한 column을 명시합니다

```
SELECT enrollment_id, member_id, course_id, status
FROM enrollment;
```

`SELECT *`는 초기 탐색에는 편하지만 계약이 약합니다.

| `SELECT *` 위험 | 영향 |
|---|---|
| schema에 column 추가 | API·export 결과가 예고 없이 바뀐다 |
| 필요 없는 대용량·민감 column | I/O·보안 범위가 커진다 |
| JOIN에서 같은 column name | 소유 표가 불명확하다 |
| ordinal에 의존하는 client | column 순서 변경에 깨진다 |

## 4.2. alias로 출력 의미를 드러냅니다

```
SELECT
  e.enrollment_id,
  m.name AS member_name,
  c.title AS course_title,
  e.status AS enrollment_status
FROM enrollment AS e
JOIN member AS m ON m.member_id = e.member_id
JOIN course AS c ON c.course_id = e.course_id;
```

table alias `e`·`m`·`c`는 짧게 쓰되 JOIN이 많을 때도 source를 쉽게 추적할 수 있어야 합니다. output alias는 사용자·API·report에서 의미가 보이도록 씁니다.

### 4.3. expression에도 NULL·type 계약이 있습니다

```
SELECT
  enrollment_id,
  COALESCE(completion_percent, 0) AS display_progress
FROM enrollment;
```

`COALESCE(..., 0)`는 표시 규칙일 수 있지만 저장된 NULL의 의미를 0으로 바꾸어 설명하는 것은 아닙니다. 집계·업무 판단에서는 "미정"과 "0%"를 따로 취급해야 할 수 있습니다.

## 5. WHERE의 TRUE·FALSE·UNKNOWN을 row별로 추적합니다

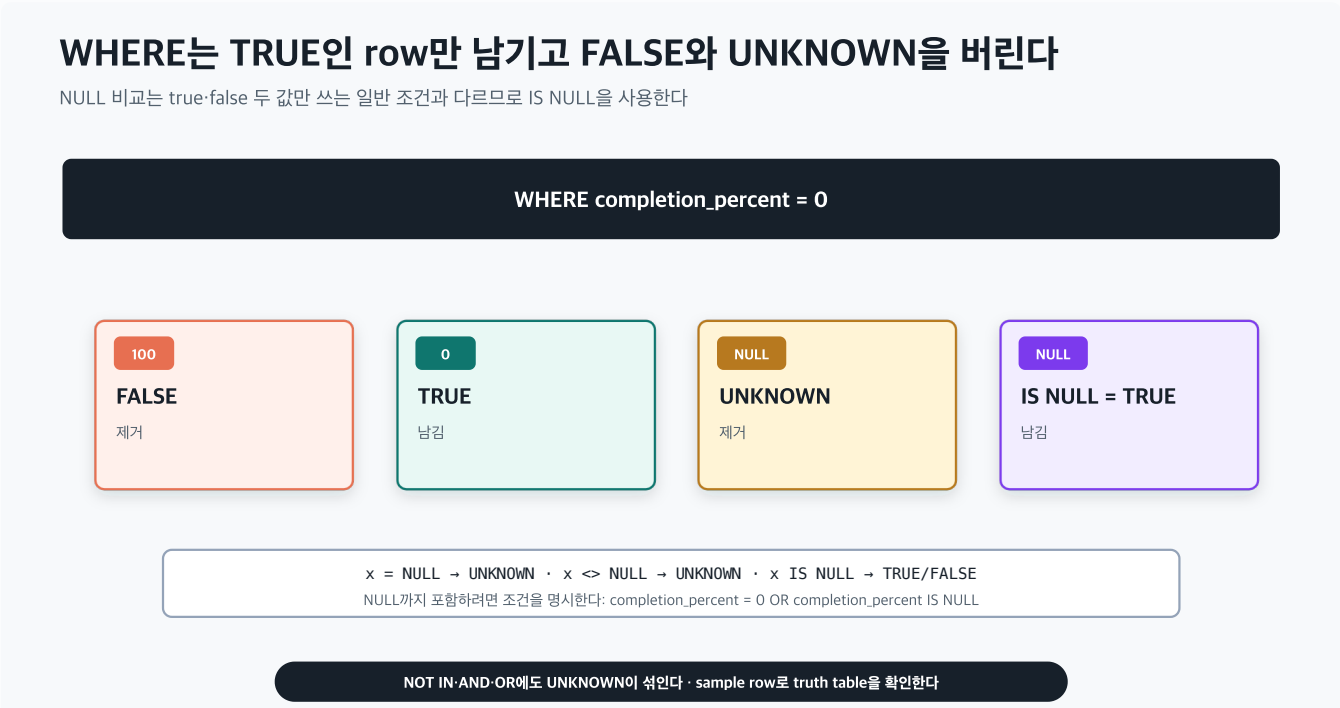



그림 4. WHERE는 TRUE인 row만 남깁니다. FALSE와 UNKNOWN은 모두 결과에서 제거됩니다.

## WHERE complet

100

**FALSE**

제거

0

**TRUE**

남김

`x = NULL → UNKNOWN` · `x <> NULL →`

NULL까지 포함하려면 조건을 명시한다: completic

## tion_percent = 0

NULL

**UNKNOWN**

제거

NULL

**IS NULL = TRUE**

남김

`UNKNOWN · x IS NULL → TRUE/FALSE`

on_percent = 0 OR completion_percent IS NULL

## 5.1. 비교와 NULL

```
-- 예상 0 rows
SELECT enrollment_id
FROM enrollment
WHERE completion_percent = NULL;

-- 예상 1 row: enr_2
SELECT enrollment_id
FROM enrollment
WHERE completion_percent IS NULL;
```

ordinary comparison에 NULL이 섞이면 결과는 보통 UNKNOWN입니다. `= NULL`, `<> NULL`로 NULL을 찾지 않습니다. `IS NULL`, `IS NOT NULL`을 사용합니다.

PostgreSQL의 `IS DISTINCT FROM` · `IS NOT DISTINCT FROM`은 NULL을 "unknown"으로 흘려보내지 않고 비교 가능한 값처럼 다룰 때 유용합니다.

```
SELECT NULL IS DISTINCT FROM NULL; -- false
SELECT 0 IS DISTINCT FROM NULL;    -- true
```

## 5.2. AND·OR에는 괄호를 씁니다

```
-- 의도: active이면서 progress가 NULL 또는 0
WHERE status = 'active'
  AND (completion_percent IS NULL OR completion_percent = 0)
```

괄호가 없으면 operator precedence에 따라 의도와 다른 row가 포함될 수 있습니다. 복합 조건은 자연어로 먼저 적고 행렬표로 검증합니다.

| status | progress | 의도한 결과 |
|---|---|---|
| active | NULL | 포함 |
| active | 0 | 포함 |
| active | 80 | 제외 |
| canceled | NULL | 제외 |

## 5.3. `NOT IN`과 NULL을 주의합니다

```
WHERE member_id NOT IN (SELECT mentor_id FROM member)
```

subquery에 NULL이 포함되면 비교가 UNKNOWN에 빠져 예상과 다른 0 rows가 될 수 있습니다. NULL 의미를 명시하고 `NOT EXISTS`로 관계 부재를 표현하는 대안을 검토합니다.

```
SELECT m.member_id
FROM member AS m
WHERE NOT EXISTS (
  SELECT 1
  FROM member AS mentee
  WHERE mentee.mentor_id = m.member_id
);
```

**30초 확인 3**
`completion_percent = 0 OR completion_percent IS NULL`은 0과 미정을 같은 업무 상태로 다루는 것입니까? 그 합친 의미를 제품 용어로 쓰세요.

# 6. JOIN은 관계와 unmatched row 보존을 함께 결정합니다

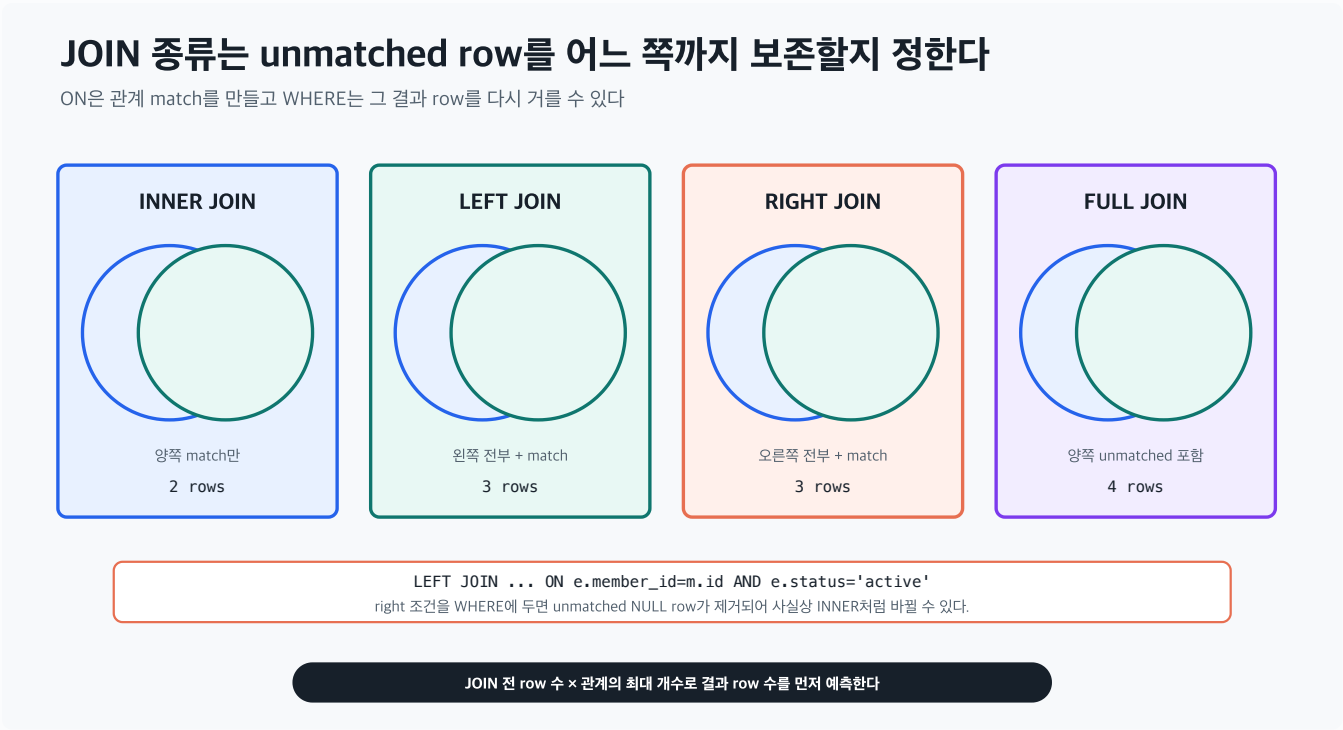
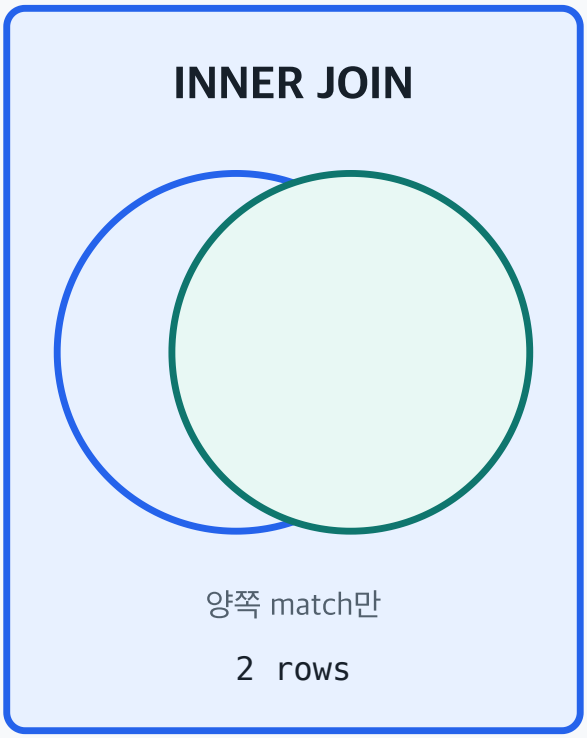
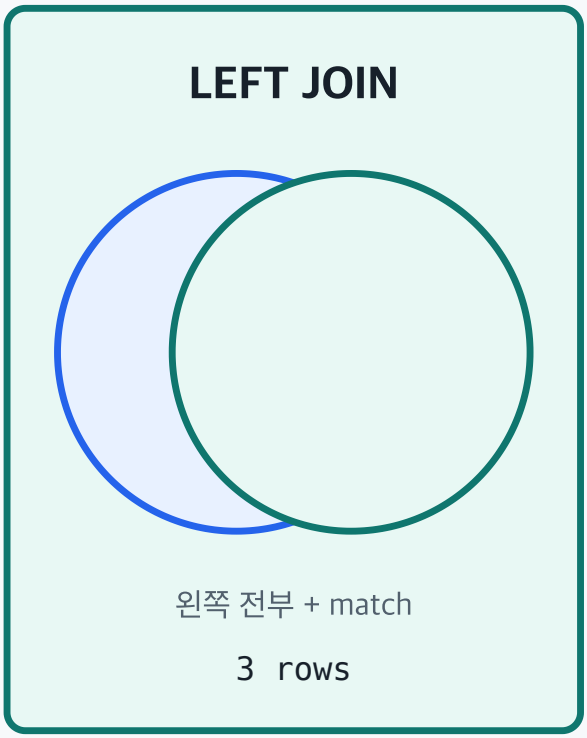



그림 5. JOIN 종류는 어느 쪽 unmatched row를 보존할지 정합니다. 그런 뒤 WHERE가 이 row를 다시 거를 수 있습니다.

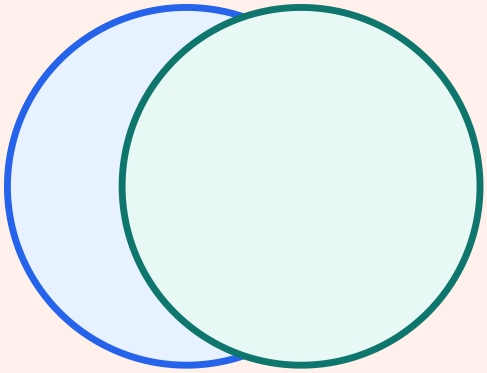



```
LEFT JOIN ... ON e.member_ic
```

right 조건을 WHERE에 두면 unmatched NULL

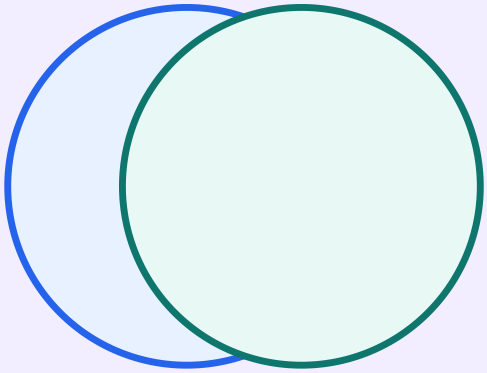
### RIGHT JOIN



오른쪽 전부 + match

3 rows

### FULL JOIN



양쪽 unmatched 포함

4 rows

```
d=m.id AND e.status='active'
```

row가 제거되어 사실상 INNER처럼 바뀔 수 있다.

## 6.1. INNER JOIN · match만 남기기

```
SELECT e.enrollment_id, m.name, c.title
FROM enrollment AS e
JOIN member AS m ON m.member_id = e.member_id
JOIN course AS c ON c.course_id = e.course_id;
```

foreign key가 유효한 학습 dataset에서 enrollment 4 rows는 모두 member·course와 match합니다. 결과는 4 rows입니다.

## 6.2. LEFT JOIN · 왼쪽 row 보존

```
SELECT m.member_id, m.name, e.enrollment_id, e.status
FROM member AS m
LEFT JOIN enrollment AS e
  ON e.member_id = m.member_id
 AND e.status = 'active'
ORDER BY m.member_id;
```

| member | active match | 결과 row |
|---|---|---|
| 윤아 | enr_1 | 윤아, enr_1 |
| 민준 | enr_3 | 민준, enr_3 |
| 서현 | 없음 | 서현, NULL |

결과는 3 rows입니다.

## 6.3. right 조건을 WHERE에 두면 unmatched row가 사라집니다

```
SELECT m.member_id, m.name, e.enrollment_id, e.status
FROM member AS m
LEFT JOIN enrollment AS e ON e.member_id = m.member_id
WHERE e.status = 'active';
```

서현의 `e.status`는 NULL입니다. `NULL = 'active'`는 UNKNOWN이므로 WHERE에서 제거됩니다. 결과는 2 rows로 줄어듭니다.

## ON과 WHERE를 나누는 질문

| 조건 | 놓을 곳 |
| --- | --- |
| 어떤 row가 관계로 match하는가 | `ON` |
| JOIN 결과 중 어떤 row를 최종 남길 것인가 | `WHERE` |
| unmatched를 보존하면서 right 상태만 match할 것인가 | `ON` |
| unmatched도 제외할 것인가 | `WHERE` 가능 |

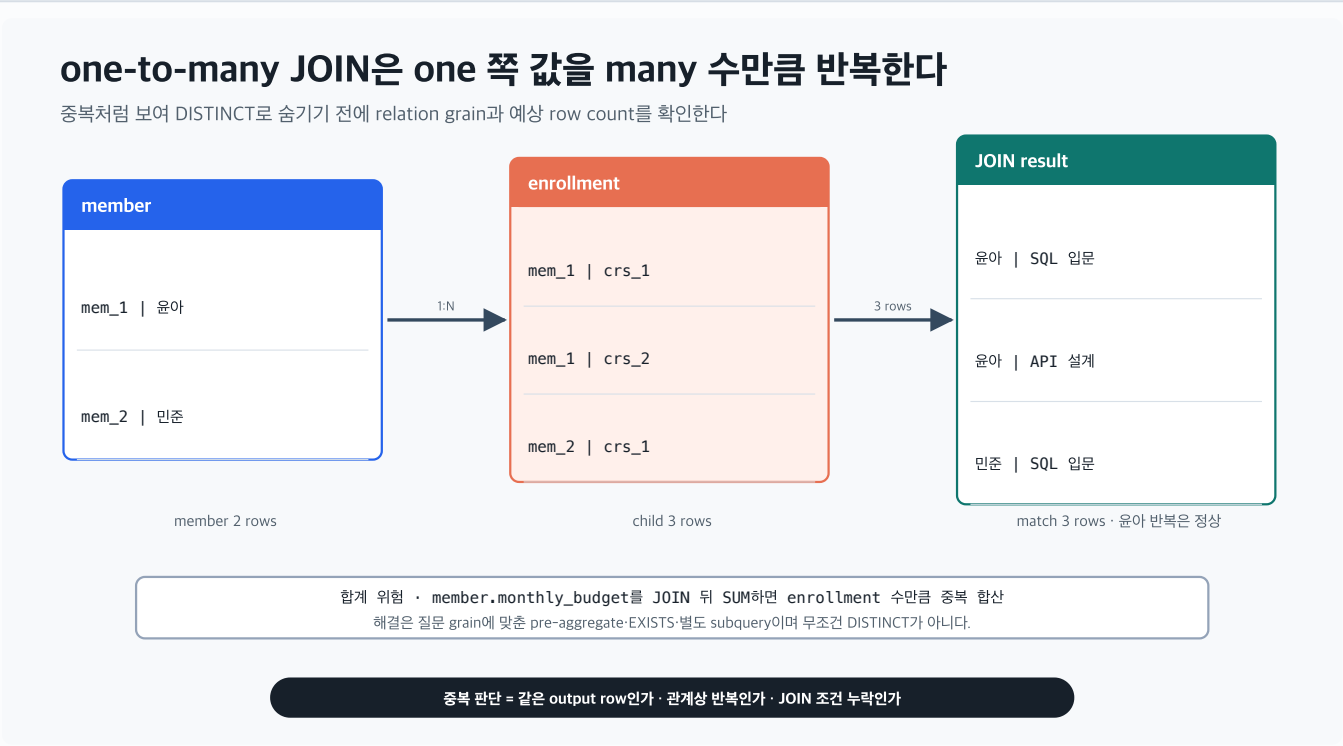
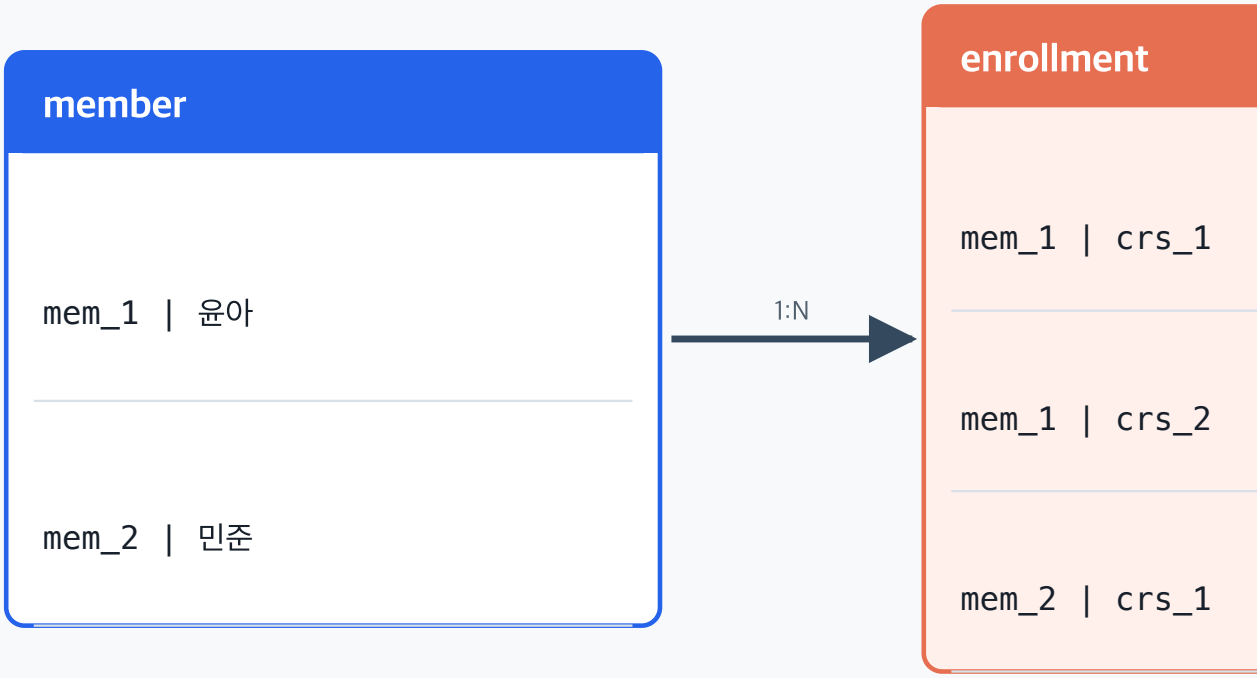



그림 6. one-to-many JOIN의 반복은 중복 오류가 아닐 수 있습니다. 문제는 one 쪽 금액·수량을 JOIN 뒤에 합산해 배수로 계산하는 경우입니다.




member 2 rows

child 3

합계 위험 · member.monthly_budget를 JO

해결은 질문 grain에 맞춘 pre-aggregate·EXISTS


IN 뒤 SUM하면 enrollment 수만큼 중복 합산

·별도 subquery이며 무조건 DISTINCT가 아니다.

## 6.4. row 증폭을 예상합니다

```
member mem_1 → enrollment 2 rows → 결과에 윤아 2번
member mem_2 → enrollment 2 rows → 결과에 민준 2번
member mem_3 → enrollment 0 rows → INNER JOIN 결과에 없음
```

## `DISTINCT`로 원인을 숨기지 않습니다

| 반복 원인 | 올바른 대응 |
| --- | --- |
| one-to-many의 정상 반복 | result grain을 many로 설명 |
| JOIN key 누락 | ON에 전체 key·scope 추가 |
| many 쪽 건수만 필요 | JOIN 대신 `EXISTS` 검토 |
| many 쪽을 요약해야 함 | 먼저 GROUP BY한 subquery와 JOIN |
| output column이 정말 동일 | 의미를 확인한 뒤 `DISTINCT` |

```sql
-- 신청이 하나라도 있는 회원이 필요하면
SELECT m.member_id, m.name
FROM member AS m
WHERE EXISTS (
  SELECT 1
  FROM enrollment AS e
  WHERE e.member_id = m.member_id
);
```

# 7. GROUP BY와 HAVING은 result grain을 바꿉니다

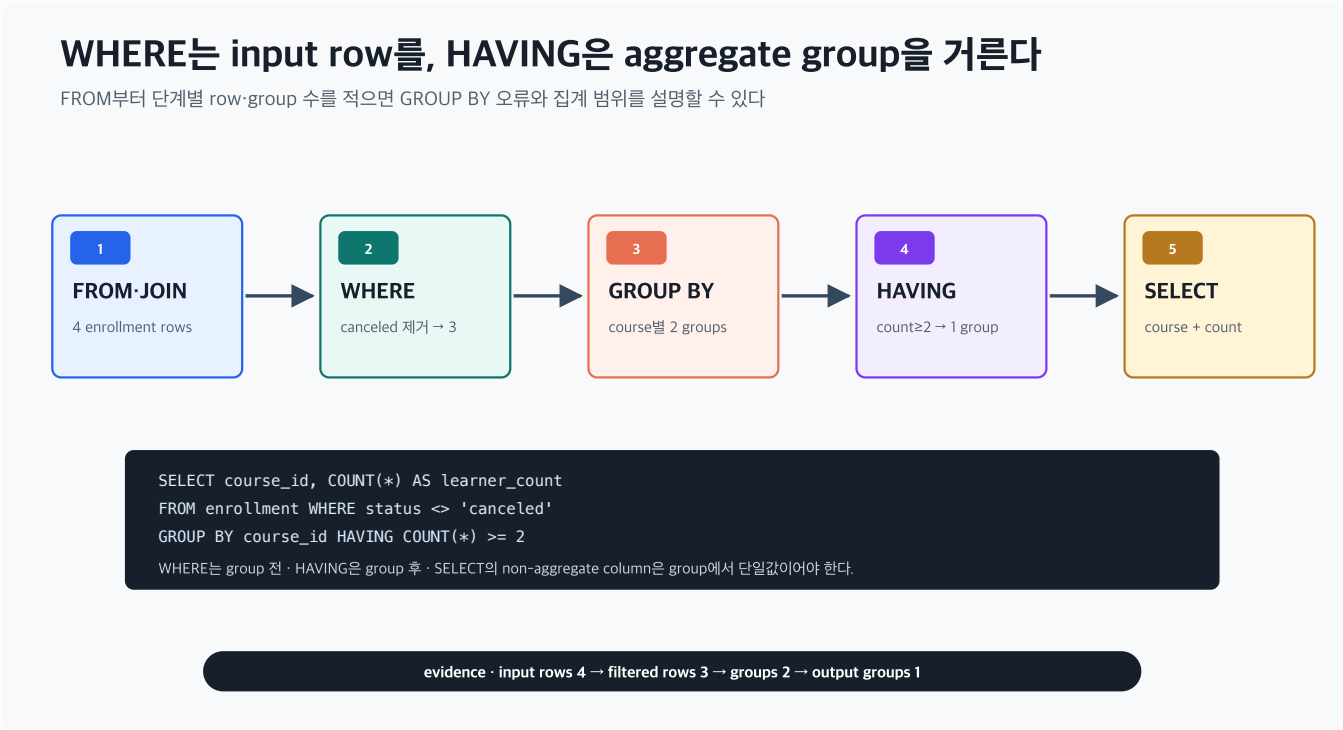



그림 7. WHERE는 집계 전 input row를, HAVING은 집계 후 group을 거릅니다.


```
SELECT course_id, COUNT(*) AS learner_count
FROM enrollment WHERE status <> 'canceled'
GROUP BY course_id HAVING COUNT(*) >= 2
```

WHERE는 group 전 · HAVING은 group 후 · SELECT의 non-aggregate column은 g


group에서 단일값이어야 한다.

## 7.1. group 전·후를 나눕니다

```
SELECT course_id, COUNT(*) AS learner_count
FROM enrollment
WHERE status <> 'canceled'
GROUP BY course_id
HAVING COUNT(*) >= 2
ORDER BY course_id;
```

| 단계 | grain | count |
|---|---|---|
| FROM enrollment | 한 row = 한 신청 | 4 rows |
| WHERE status <> canceled | 한 row = 한 신청 | 3 rows |
| GROUP BY course_id | 한 group = 한 강좌 | 2 groups |
| HAVING COUNT(*) >= 2 | 한 group = 한 강좌 | 1 group |
| SELECT | 한 row = 한 강좌 집계 | 1 row |

## 7.2. SELECT list의 non-aggregate column은 group key여야 합니다

```
-- name을 어떤 row에서 고를지 정의할 수 없다
SELECT course_id, member_id, COUNT(*)
FROM enrollment
GROUP BY course_id;
```

PostgreSQL은 일반적으로 group key가 아니고 aggregate도 아닌 column을 허용하지 않습니다. functional dependency가 인식되는 특정 경우는 예외가 있지만 초급 설계에서는 “output grain을 구성하는 key인가”를 먼저 확인합니다.

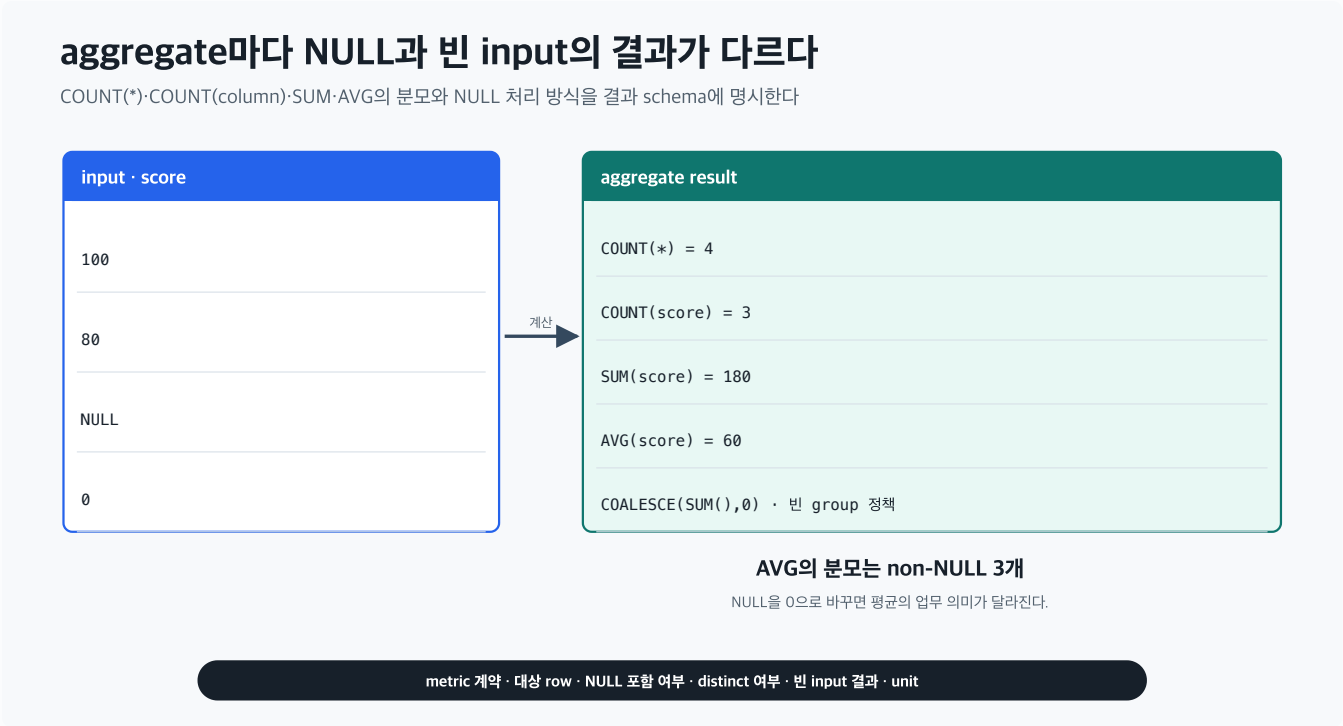
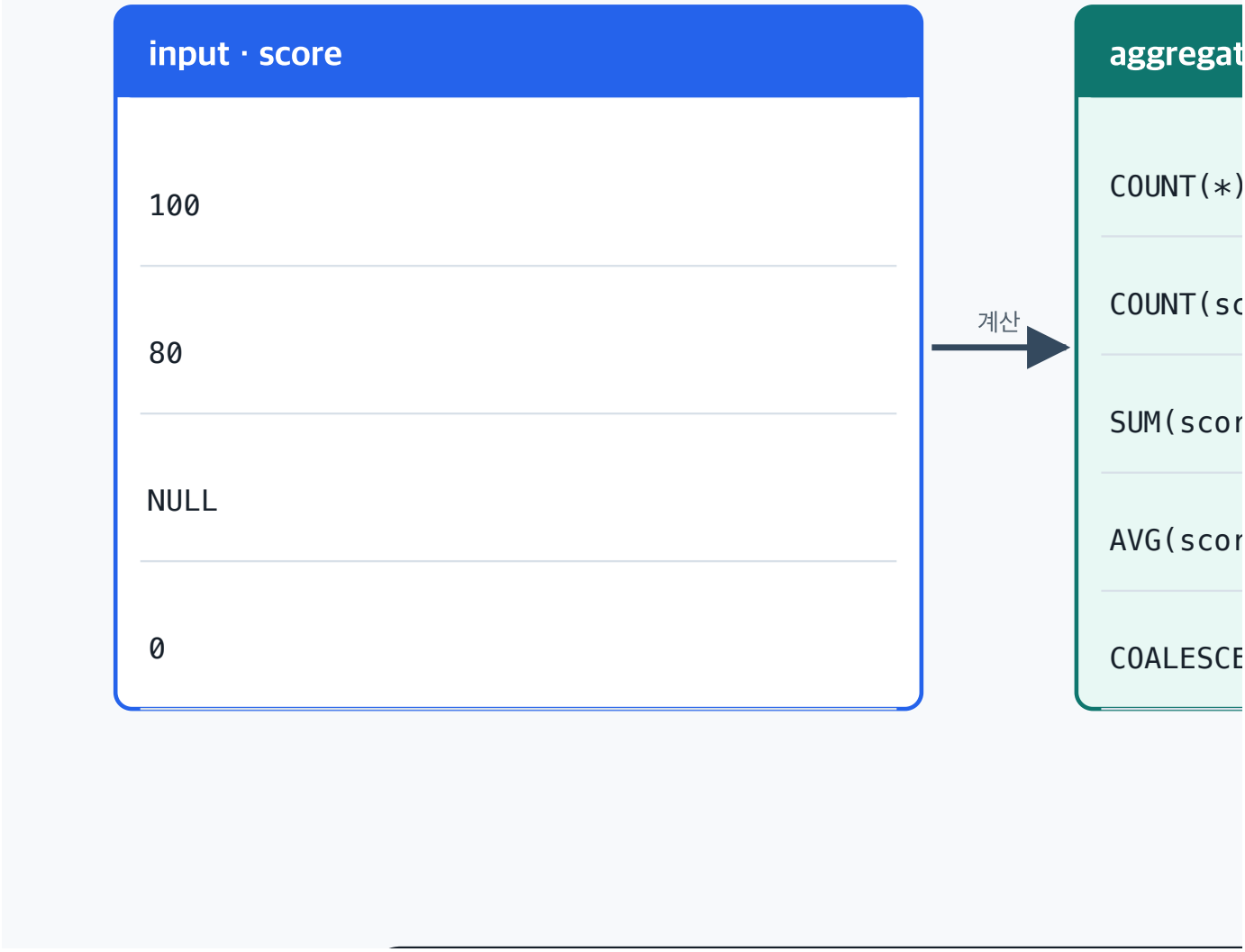



그림 8. aggregate마다 NULL을 세는 방법이 다릅니다. metric 정의서에 분자·분모·NULL 규칙을 써야 합니다.


input · score
100
80
NULL
0
계산
aggregat
COUNT(*)
COUNT(sc
SUM(scor
AVG(scor
COALESCE

te result

```
 = 4
core) = 3
re) = 180
re) = 60
E(SUM(),0) · 빈 group 정책
```

## AVG의 분모는 non-NULL 3개

NULL을 0으로 바꾸면 평균의 업무 의미가 달라진다.

## 7.3. COUNT와 NULL

```
SELECT
  COUNT(*) AS all_rows,
  COUNT(completion_percent) AS known_progress,
  SUM(completion_percent) AS progress_sum,
  AVG(completion_percent) AS progress_avg
FROM enrollment;
```

| expression | 결과 | 의미 |
|---|---|---|
| `COUNT(*)` | 4 | input row 전체 |
| `COUNT(completion_percent)` | 3 | non-NULL 값 개수 |
| `SUM(completion_percent)` | 180 | non-NULL 100+80+0 |
| `AVG(completion_percent)` | 60 | 180 / non-NULL 3 |

## 7.4. metric 계약

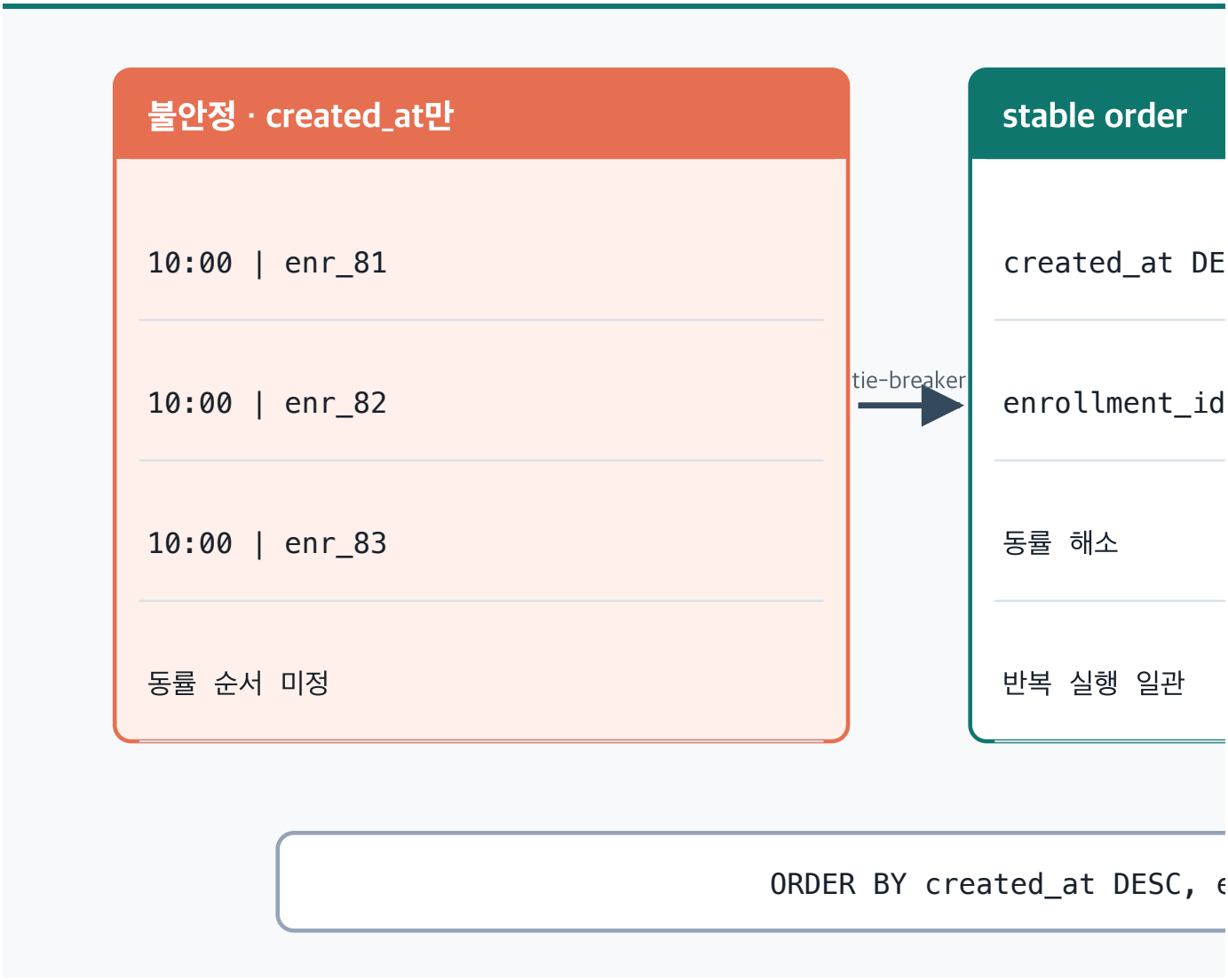
```
진도 평균 = completion_percent가 NULL이 아닌 enrollment의 합 / non-NULL 건수
대상 상태 = active + completed
기간 기준 = created_at [start, end)
소수·rounding = 1 decimal, half up
```

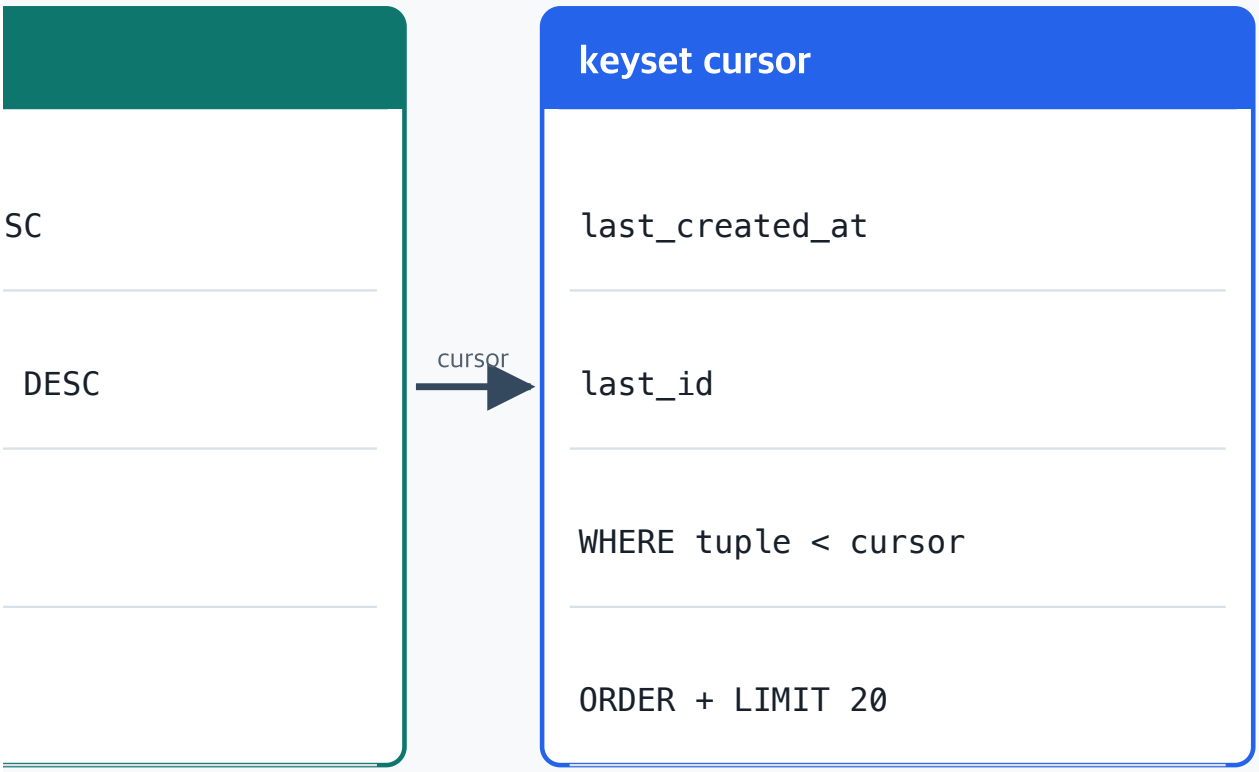
`COALESCE(completion_percent, 0)`를 AVG 안에 넣으면 NULL이 0으로 바뀌어 분모가 4가 됩니다. 이것이 업무 의미와 맞는지 확인합니다.

**30초 확인 4**
미정 진도를 0으로 간주하면 평균은 45가 됩니다. 60과 45 중 어느 것이 올바른지는 SQL이 아니라 어떤 계약이 결정합니까?

# 8. ORDER BY없이 안정적인 LIMIT은 없습니다


그림 9. 동률을 해소하는 stable key가 있어야 어느 row가 먼저인지 일의적으로 정할 수 있습니다.


불안정 · created_at만
10:00 | enr_81
10:00 | enr_82
10:00 | enr_83
동률 순서 미정
tie-breaker
stable order
created_at DE
enrollment_id
동률 해소
반복 실행 일관
ORDER BY created_at DESC, e




```
enrollment_id DESC LIMIT 20
```

## 8.1. 명시적 ORDER BY

PostgreSQL 문서는 ORDER BY가 없으면 결과 row의 순서가 예측 불가능하다고 설명합니다. query plan·disk layout·parallelism·index가 바뀌면 순서도 바뀌 수 있습니다.

```
SELECT enrollment_id, created_at, status
FROM enrollment
ORDER BY created_at DESC, enrollment_id DESC
LIMIT 20;
```

`created_at`이 같은 enr_1·enr_2는 `enrollment_id`로 다시 순서를 정합니다.

## 8.2. OFFSET의 의미

```
ORDER BY created_at DESC, enrollment_id DESC
LIMIT 20 OFFSET 40;
```

OFFSET은 간단하지만 큰 offset에서 앞 row를 건너뛰는 비용이 커질 수 있고, 페이지 이동 사이에 row가 추가·삭제되면 누락·중복이 생길 수 있습니다.

## 8.3. keyset cursor 개념

PostgreSQL tuple comparison 예:

```
SELECT enrollment_id, created_at, status
FROM enrollment
WHERE (created_at, enrollment_id) < ($1, $2)
ORDER BY created_at DESC, enrollment_id DESC
LIMIT 20;
```

| cursor 항목 | 계약 |
| --- | --- |
| ordered fields | `created_at DESC, enrollment_id DESC` |
| cursor values | 마지막 row의 두 값 |
| NULL | order·comparison 규칙 명시 |
| encoding | 변조 방지·version 검토 |

| cursor 항목 | 계약 |
| --- | --- |
| 동시 변경 | snapshot 요구 또는 허용 오차 정의 |

# 9. parameterized query로 SQL code와 data를 분리합니다

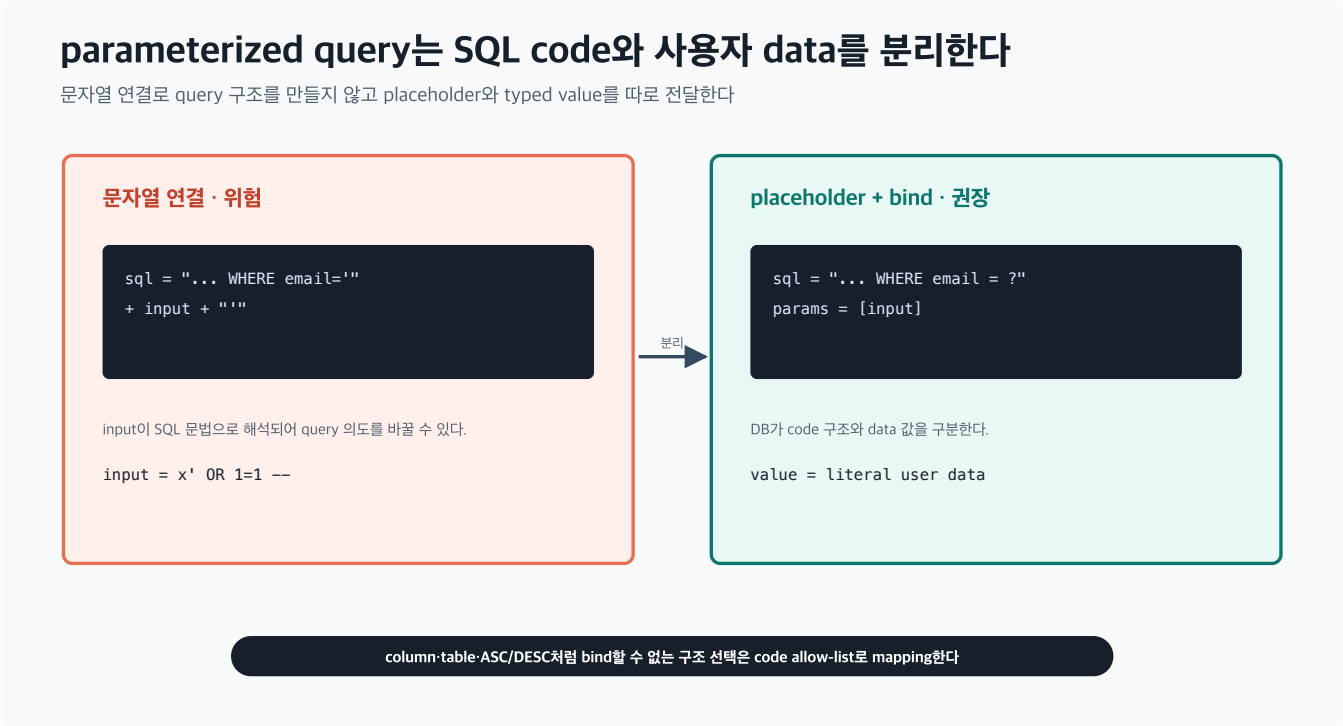



그림 10. parameter binding은 사용자 값을 SQL 문법이 아니라 data로 전달합니다. 이것은 정확성·보안·type 계약의 기본입니다.

## 문자열 연결 · 위험

```
sql = "... WHERE email='"
+ input + "'"
```

input이 SQL 문법으로 해석되어 query 의도를 바꿀 수 있다.

```
input = x' OR 1=1 --
```

분

column·table·ASC/DESC처럼 bind할 수 없는

## placeholder + bind · 권장

```
sql = "... WHERE email = ?"
params = [input]
```

리

DB가 code 구조와 data 값을 구분한다.

`value = literal user data`

구조 선택을 code allow-list로 mapping한다

## 9.1. 문자열 연결을 사용하지 않습니다

```
// 위험: input이 SQL 구조를 바꿀 수 있다
const sql = "SELECT * FROM member WHERE email = '" + input + "'";
```

OWASP는 SQL injection 방어의 주요 방법으로 prepared statement·parameterized query를 권고합니다. DB driver에 SQL code와 parameter를 따로 전달합니다.

PostgreSQL client 개념 예:

```
SELECT enrollment_id, course_id, status
FROM enrollment
WHERE member_id = $1 AND status = $2
ORDER BY enrollment_id;
```

```
parameters = ["mem_1", "active"]
```

실습기의 SQLite/sql.js는 `?` placeholder를 사용합니다.

```
WHERE member_id = ? AND status = ?
```

## 9.2. 모든 것을 parameter로 bind할 수는 없습니다

table name, column name, `ASC`·`DESC` 같은 SQL 구조는 보통 value parameter로 bind할 수 없습니다. 사용자가 선택해야 한다면 코드 allow-list로 안전한 SQL 토큰에 mapping합니다.

```
const sortMap = {
  newest: 'created_at DESC, enrollment_id DESC',
  oldest: 'created_at ASC, enrollment_id ASC'
};
const orderBy = sortMap[userChoice] ?? sortMap.newest;
```

### 9.3. parameter 증거

| 항목 | 저장 여부 | 주의 |
| --- | --- | --- |
| SQL template·hash | 필수 | 문장 버전 식별 |
| parameter name·type | 필수 | 타입·NULL 계약 |
| 민감 원문 | 최소화 | password·token·주민번호 log 금지 |
| parameter value hash·mask | 선택 | 재현성과 보안 균형 |
| actor·tenant | 필수 | scope 검증 |

**30초 확인 5**
사용자가 “최신순·오래된순”을 고를 때 `ORDER BY ?` 하나로 처리하지 않고 allow-list mapping을 쓰는 이유는 무엇입니까?

## 10. 조회는 result schema·count·sample로 검증합니다

### 10.1. query evidence 최소 단위

```
question       : active 신청의 ID·회원·강좌를 본다
result grain   : 한 row = 한 active enrollment
SQL hash       : query.active-enrollment.v3
parameters     : tenant_id=tnt_1, status=active
expected rows  : 2
actual rows    : 2
result schema  : enrollment_id text, member_id text, course_id text
sample IDs     : enr_1, enr_3
NULL count     : 0 / 3 output columns
ordered by     : enrollment_id ASC
```

### 10.2. 0 rows도 설명합니다

0 rows는 다음 여러 의미일 수 있습니다.

| 원인 | 확인 |
| --- | --- |
| 정상적으로 data가 없음 | 업무 현황·test fixture |

| 원인 | 확인 |
| --- | --- |
| tenant·상태·기간 scope가 다름 | parameter 값 |
| NULL 비교 실수 | `= NULL` · `NOT IN` |
| INNER JOIN이 unmatched를 제거 | JOIN 종류·ON |
| LEFT JOIN 후 WHERE가 NULL row 제거 | logical pipeline |
| 권한 정책이 row를 필터링 | actor·policy context |

## 10.3. sample만 보지 않습니다

sample 5 rows가 그럴듯해도 나머지 100만 rows가 틀릴 수 있습니다.
count·min/max·NULL·distinct·duplicate·unmatched 지표를 함께 봅니다.

```
SELECT
  COUNT(*) AS row_count,
  COUNT(DISTINCT enrollment_id) AS distinct_ids,
  COUNT(*) FILTER (WHERE completion_percent IS NULL) AS null_progress
FROM enrollment;
```

`FILTER` clause는 PostgreSQL 문법입니다. 실습기 SQLite에서도 현재 버전은 지원하지만 운영 DB의 버전·dialect를 확인합니다.

# 11. INSERT·UPDATE·DELETE는 대상 범위 증거를 먼저 만듭니다

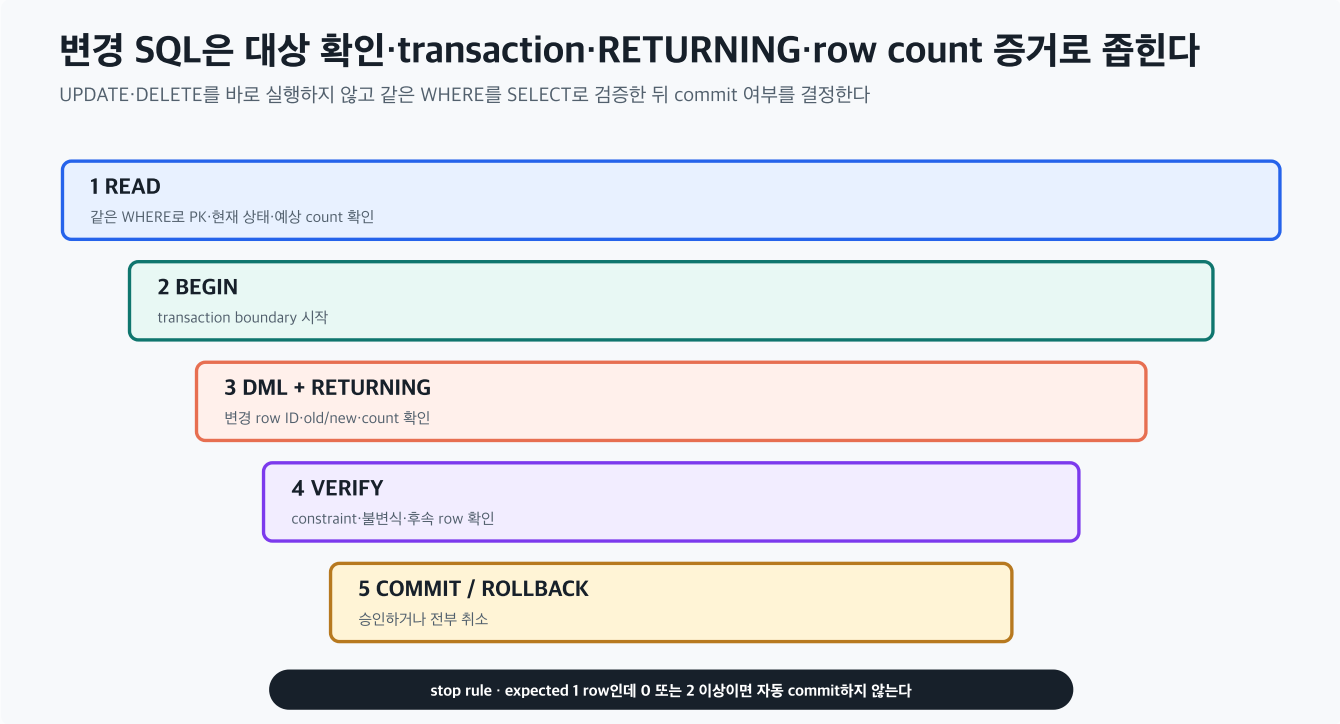
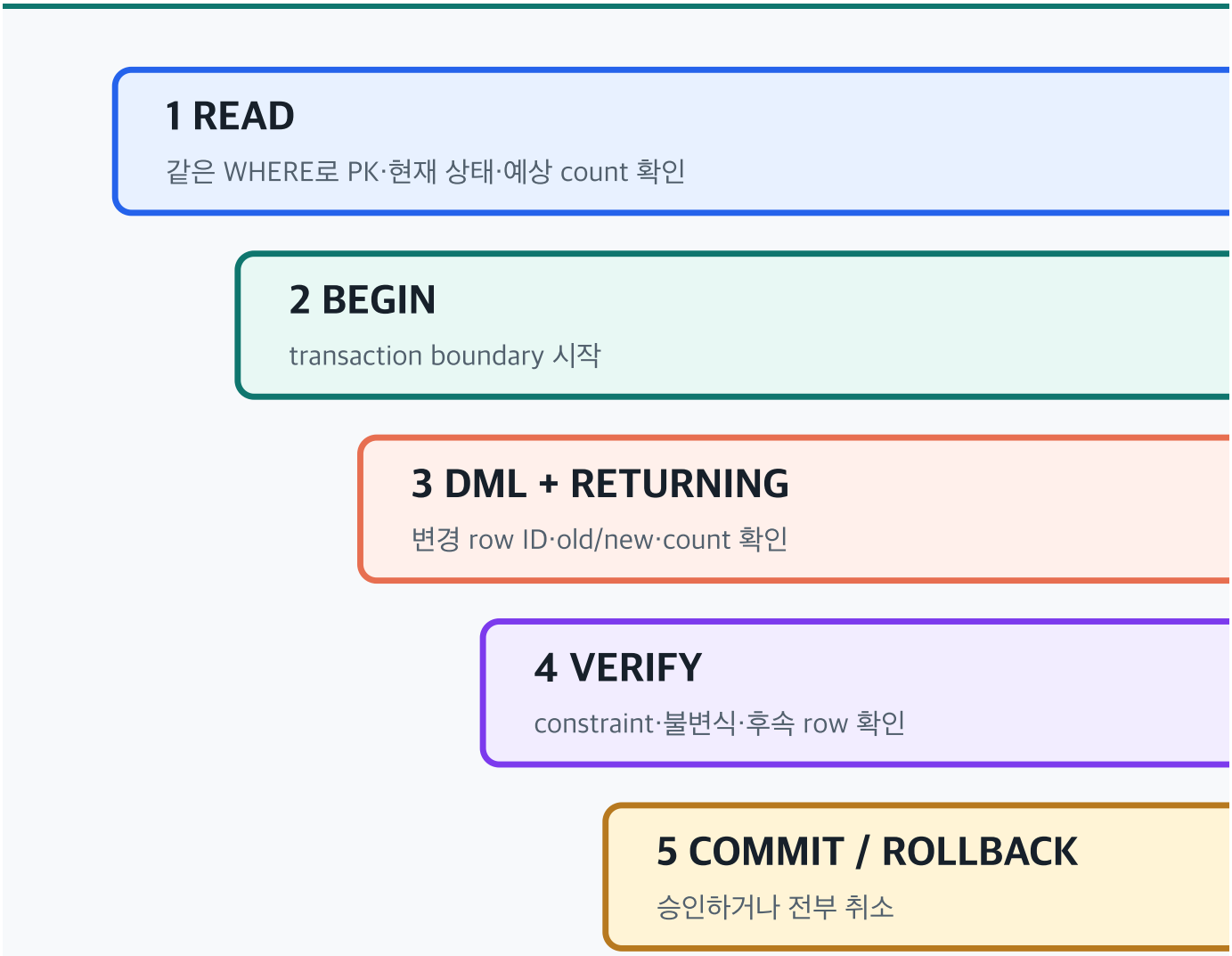
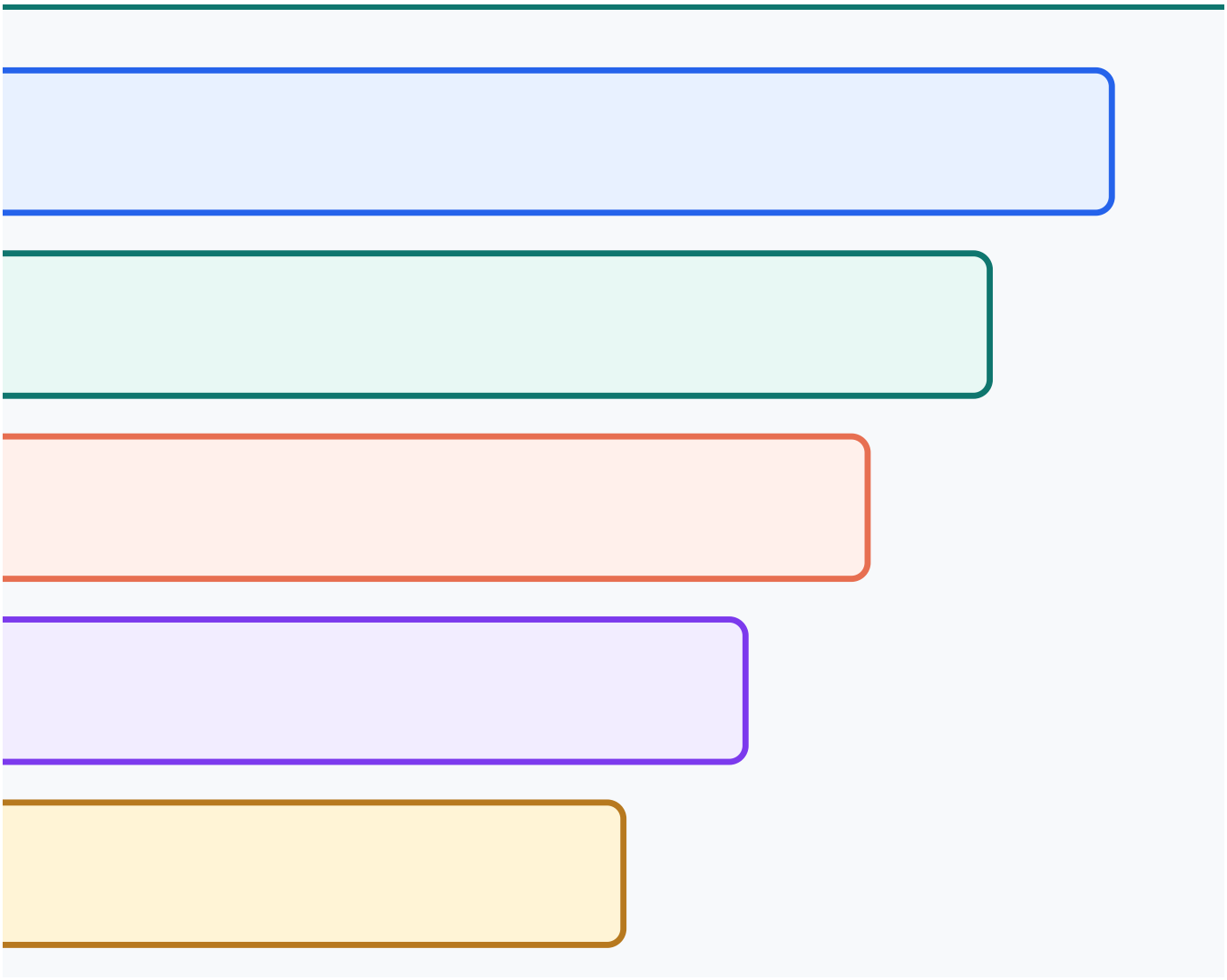



그림 11. 변경 SQL은 바로 실행하지 않고 대상·transaction·반환 row·불변식을 순서대로 검증합니다.


1 READ
같은 WHERE로 PK·현재 상태·예상 count 확인
2 BEGIN
transaction boundary 시작
3 DML + RETURNING
변경 row ID·old/new·count 확인
4 VERIFY
constraint·불변식·후속 row 확인
5 COMMIT / ROLLBACK
승인하거나 전부 취소



## 11.1. INSERT · column을 명시합니다

PostgreSQL parameter 개념 예:

```
INSERT INTO enrollment (
  enrollment_id,
  member_id,
  course_id,
  status,
  completion_percent,
  created_at
) VALUES ($1, $2, $3, $4, $5, $6)
RETURNING enrollment_id, member_id, course_id, status;
```

| 검수 | 질문 |
|---|---|
| column 명시 | schema 순서와 독립적입니까 |
| parameter | code와 data가 분리되었습니까 |
| key | client·API·DB 중 생성 주체가 분명합니까 |
| defaults | 생략의 의미가 schema와 맞습니까 |
| constraint | FK·UNIQUE·CHECK 실패를 사용자 결과로 바꾸었습니까 |
| RETURNING | 실제 생성된 ID·default·status를 받았습니까 |

## 11.2. UPDATE · 현재 상태를 precondition으로 씁니다

먼저 같은 WHERE의 target SELECT를 실행합니다.

```
SELECT enrollment_id, status, completion_percent
FROM enrollment
WHERE tenant_id = $1
  AND enrollment_id = $2
  AND status = 'active';
```

예상 count가 1일 때만 변경을 진행합니다.

```
UPDATE enrollment
SET status = 'completed'
WHERE tenant_id = $1
  AND enrollment_id = $2
  AND status = 'active'
RETURNING enrollment_id, status;
```

| actual count | 해석 | 다음 행동 |
|---|---|---|
| 0 | not found·wrong tenant·stale state | commit 금지, 상태 재조회 |
| 1 | expected target | 불변식 검증 후 commit |
| 2 이상 | key·scope·join 오류 | rollback, SQL 재설계 |

## 11.3. DELETE · 삭제는 보존 정책을 포함합니다

```
DELETE FROM enrollment
WHERE tenant_id = $1
  AND enrollment_id = $2
  AND status = 'canceled'
RETURNING enrollment_id, member_id, course_id, status;
```

SQL 범위가 1 row여도 hard delete가 업무·법정·감사 정책에 맞는지는 별도로 확인합니다. soft delete·status transition·archive·anonymization을 검토할 수 있습니다.


그림 12. 변경 위험은 SQL 길이가 아니라 해당 문장이 만질 수 있는 row 범위입니다.

WHERE 없음

## all rows

즉시 중단

tenant 누락

## cross-tenant

scope 추가

```
SELECT enrollment_id, status FROM enrollment
WHERE tenant_id = ? AND enrollment_id = ? AND status = 'acti

UPDATE enrollment SET status='canceled' WHERE ... RETURNING
```

expected count=1 · 0이면 stale state·not found · 2 이상이면 key·scope 오류

상태 조건 누락

### terminal 포함

precondition 추가

join 증폭

### same row 반복 match

EXISTS·unique join

```
ive';

 enrollment_id, status;
```

## 11.4. WHERE 없는 UPDATE·DELETE는 stop rule입니다

```
UPDATE enrollment SET status = 'canceled';
DELETE FROM enrollment;
```

PostgreSQL `DELETE` 문서는 WHERE가 없으면 table의 모든 row가 삭제된다고 명시합니다. 전체 변경이 정말 의도인 migration·maintenance라면 별도 승인·backup·count·rollback plan을 가진 전용 runbook으로 수행합니다.

## 11.5. JOIN을 사용한 변경은 match 증폭을 확인합니다

PostgreSQL은 `UPDATE ... FROM`, `DELETE ... USING`을 지원합니다. source JOIN이 target row 하나에 여러 row를 match하면 결과가 비결정적이거나 의도와 다를 수 있습니다. target PK별 match count가 1 이하인지 먼저 조회합니다.

```
SELECT target.enrollment_id, COUNT(*) AS match_count
FROM enrollment AS target
JOIN change_request AS source
  ON source.enrollment_id = target.enrollment_id
GROUP BY target.enrollment_id
HAVING COUNT(*) > 1;
```

**30초 확인 6**
`UPDATE ... WHERE enrollment_id = $1`보다 `... AND status = 'active'`를 추가한 문장이 동시 변경을 더 잘 감지하는 이유는 무엇입니까?

# 12. transaction으로 변경의 확정 경계를 만듭니다

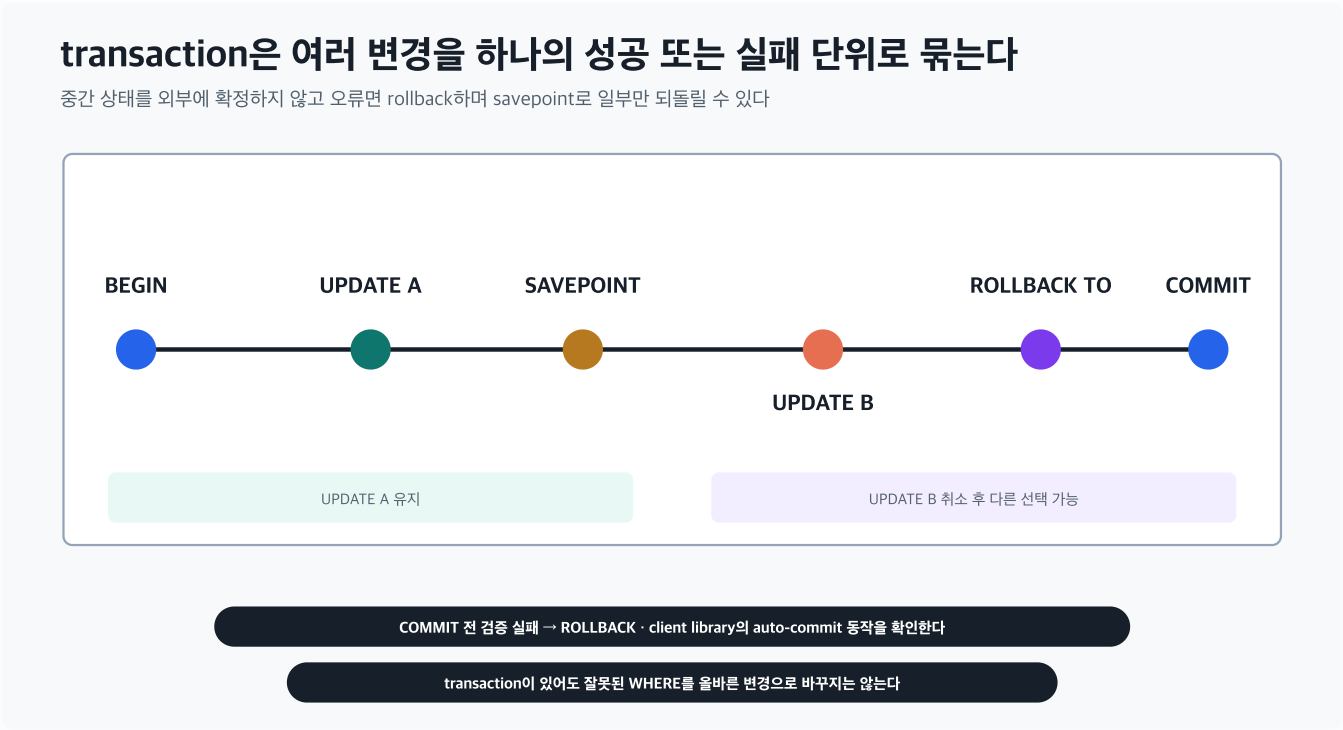
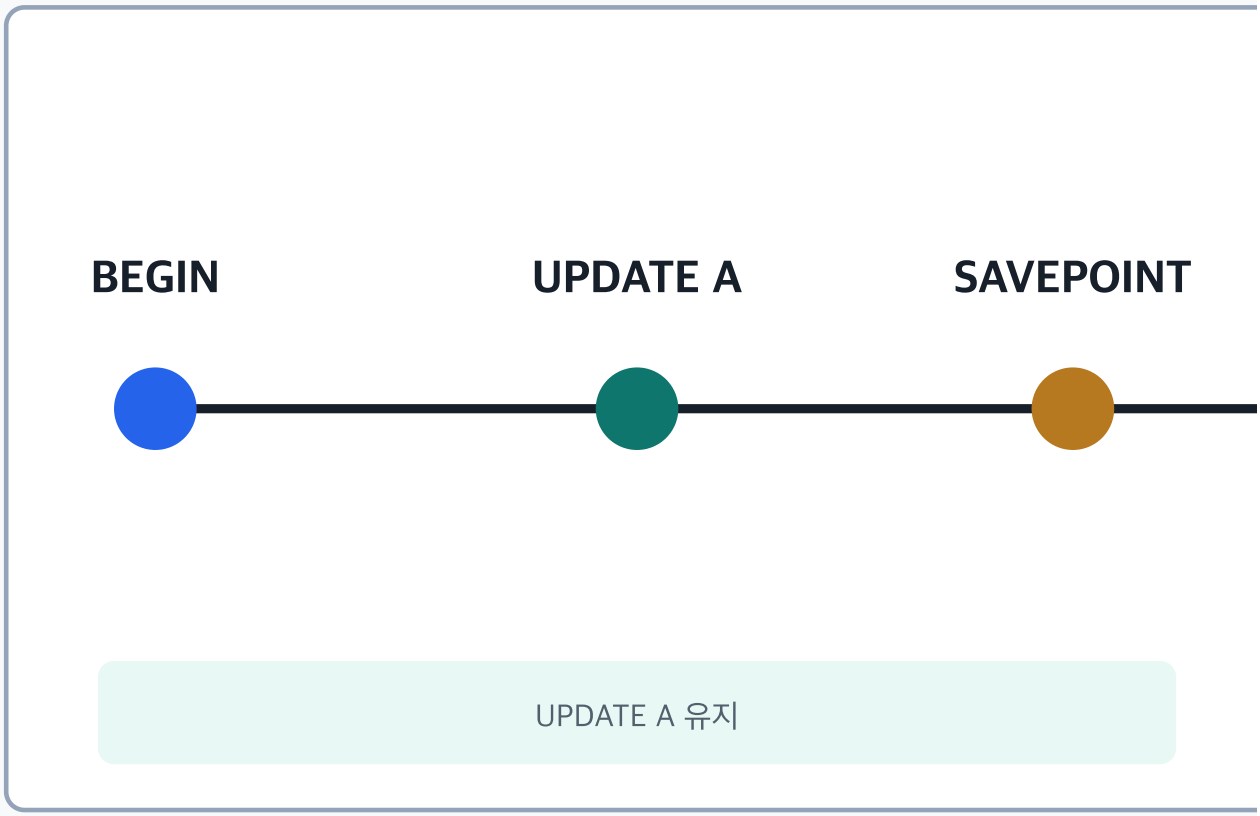
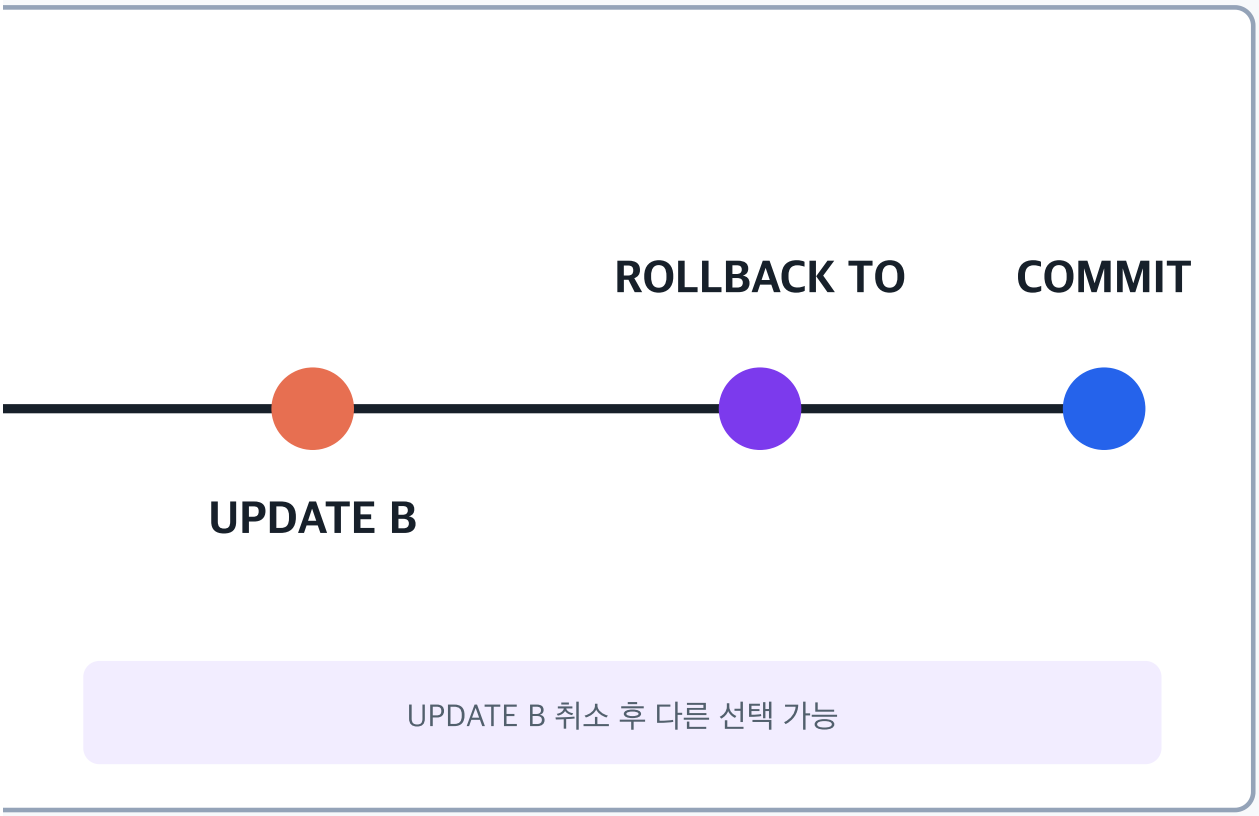



그림 13. transaction은 여러 변경을 하나의 성공·실패 단위로 묶습니다. savepoint는 그 안에서 일부 되돌림 지점을 만듭니다.




**COMMIT 전 검증 실패 → ROLLBACK · clie**




nt library의 auto-commit 동작을 확인한다

## 12.1. BEGIN·COMMIT·ROLLBACK

```
BEGIN;

UPDATE enrollment
SET status = 'completed'
WHERE enrollment_id = $1
  AND status = 'active'
RETURNING enrollment_id, status;

-- returned row·count·업무 불변식 검증
COMMIT;
```

검증이 실패하면:

```
ROLLBACK;
```

PostgreSQL 튜토리얼은 transaction을 여러 statement를 all-or-nothing 단위로 묶는 기능으로 설명합니다. 전체가 성공하면 COMMIT, 아니면 ROLLBACK입니다.

## 12.2. savepoint

```
BEGIN;

UPDATE enrollment
SET status = 'completed'
WHERE enrollment_id = 'enr_1' AND status = 'active';

SAVEPOINT after_first;

UPDATE enrollment
SET status = 'canceled'
WHERE enrollment_id = 'enr_3' AND status = 'active';

ROLLBACK TO after_first;
RELEASE after_first;
COMMIT;
```

최종 상태는 enr_1 completed, enr_3 active입니다.

### 12.3. auto-commit을 확인합니다

PostgreSQL에서 각 statement는 transaction 안에서 실행됩니다. 그러나 client·framework·DB console이 statement마다 자동 commit하는지, 명시적 BEGIN을 어떻게 다루는지는 도구별로 확인해야 합니다.

### 12.4. transaction이 해결하지 못하는 것

| transaction의 보장 | transaction이 자동 해결하지 못함 |
| --- | --- |
| commit 전 변경 취소 | 잘못된 WHERE |
| 여러 statement의 atomicity | 업무 규칙 오해 |
| 오류 시 전체 rollback | 잘못된 data를 올바른 data로 변환 |
| isolation level에 따른 concurrency 제어 | 모든 동시성 문제 제거 |
| durable commit | backup·disaster recovery 대체 |

### 12.5. stop rule

```
expected 1 row
├─ actual 0 → ROLLBACK · stale/not found/scope 재확인
├─ actual 1 → invariant 검증 후 COMMIT
└─ actual 2+ → ROLLBACK · key/join/tenant 설계 오류
```

## 13. PostgreSQL 교재와 SQLite/sql.js 실습기의 경계를 알아둡니다

이 교재의 기준 문법은 PostgreSQL 18입니다. 실습기는 브라우저에서 즉시 실행하기 위해 sql.js 1.14.1의 SQLite 3.49.1 memory database를 사용합니다. SELECT·JOIN·GROUP·DML·RETURNING·transaction의 핵심 습관은 공통이지만 dialect 차이는 사라지지 않습니다.

| 항목 | PostgreSQL 18 교재 | SQLite/sql.js 실습기 |
| --- | --- | --- |
| positional parameter | `$1`, `$2` | `?` |
| RIGHT·FULL JOIN | 지원 | 현재 SQLite도 지원하지만 실습은 INNER·LEFT 중심 |
| `RETURNING` | 지원, PostgreSQL extension 세부 기능 있음 | 현재 engine에서 지원 |

| 항목 | PostgreSQL 18 교재 | SQLite/sql.js 실습기 |
| --- | --- | --- |
| UPDATE join | `UPDATE ... FROM` | 문법·제약 다름 |
| DELETE join | `DELETE ... USING` | 문법·제약 다름 |
| type | 엄격한 type·다양한 native type | dynamic typing 특성, STRICT table 별도 |
| concurrent server | multi-client server | 브라우저 내 single memory DB 학습 |
| query plan | PostgreSQL optimizer | SQLite optimizer |

## 하지 말아야 할 추론

> 실습기에서 돌아갔다 ≠ 운영 PostgreSQL에서 같은 문법·type·plan·concurrency를 보장한다.

운영으로 옮기기 전에 선택한 DB 버전의 공식 문서, 현실 schema, 인덱스, 권한, transaction isolation, test data로 다시 검증합니다.

# 14. SQL 조회·변경 증거 스튜디오 실습

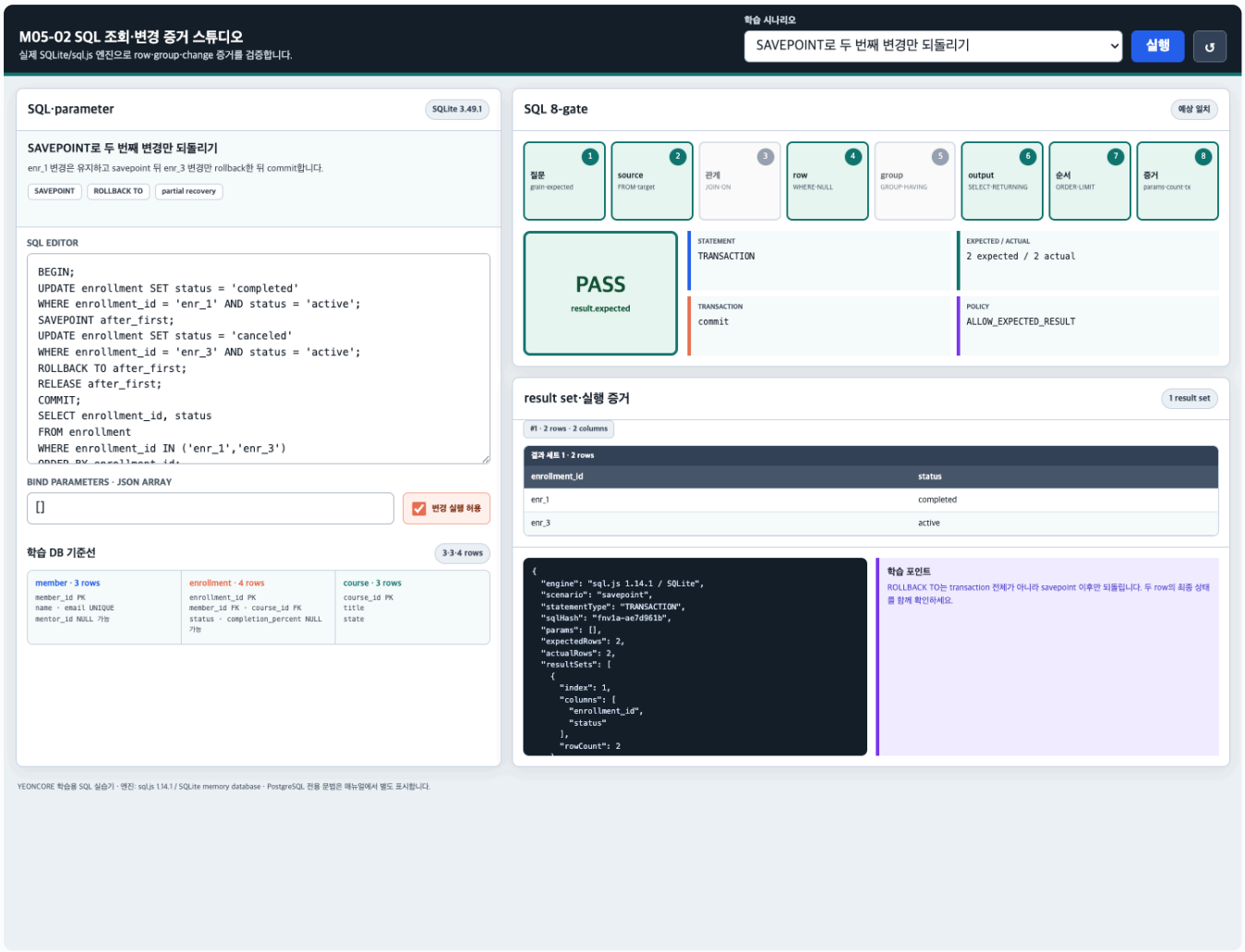



그림 14. 실습기는 편집한 SQL을 실제 SQLite 엔진에 실행합니다. expected·actual·parameter·result schema·rows modified·transaction 결과를 한 화면에서 보여 줍니다.

## 14.1. 실습 순서

1. 시나리오 제목을 읽습니다.
2. 실행 전에 expected row·group 수를 말합니다.
3. SQL과 JSON parameter를 읽습니다.
4. 조회는 바로 실행합니다.
5. 변경은 target 범위를 확인한 뒤 `변경 실행 허용`을 켭니다.
6. expected·actual, result table, JSON 증거를 비교합니다.
7. SQL 한 줄을 바꾸고 예상을 다시 말한 뒤 실행합니다.
8. 초기화 버튼으로 학습 DB를 복원합니다.

## 14.2. 20개 시나리오 지도

| 영역 | 시나리오 | 핵심 관찰 |
|---|---|---|
| 조회 | 전체, active filter | column·row·parameter |
| NULL | `= NULL` , `IS NULL` | UNKNOWN과 0 rows |
| JOIN | INNER, LEFT ON, LEFT WHERE, row 증폭 | unmatched·1:N |
| 집계 | GROUP BY, HAVING, COUNT NULL | row→group·분모 |
| 순서 | stable ORDER BY·LIMIT | tie-breaker |
| 보안 | parameterized query | code·data 분리 |
| 변경 | INSERT, safe UPDATE, broad UPDATE, DELETE | RETURNING·blast radius |
| transaction | rollback, commit, savepoint | 확정·취소 경계 |

## 14.3. 변형 미션

### 미션 A · 서현도 남기기

LEFT JOIN ON의 active 조건을 WHERE로 옮깁니다.

| 변경 전 | 변경 후 |
|---|---|
| expected 3 rows | expected 2 rows |
| 서현 + NULL 보존 | 서현 제거 |

다시 서현을 남기는 SQL을 작성합니다.

### 미션 B · 집계 분모 바꾸기

`AVG(completion_percent)`를 `AVG(COALESCE(completion_percent, 0))`으로 바꾸고 60이 45로 바뀌는 이유를 씁니다.

### 미션 C · stale update

safe UPDATE의 parameter에서 expected current status를 `completed`로 바꾸어 actual 0 row를 만듭니다. 이 0이 실패인지 idempotent success인지 API 계약을 씁니다.

### 미션 D · rollback 증거

transaction rollback 시나리오의 마지막 SELECT를 없애고 실행합니다. 어떤 증거가 사라지는지 기록한 뒤 SELECT를 복원합니다.

## 15. 조회·변경 의사결정표

| 상황 | 주요 문법 | 실행 전 증거 | 실행 후 증거 | stop rule |
|---|---|---|---|---|
| 단순 목록 | SELECT·WHERE·ORDER | grain·expected count | schema·count·sample | 예상 범위 이탈 |
| 관계 목록 | JOIN·ON | cardinality·unmatched | row 증폭·NULL | one 쪽 중복 합산 |
| 집계 | GROUP·HAVING | input row·group 수 | metric·NULL 분모 | 정의서와 불일치 |
| pagination | ORDER·LIMIT | total order·tie-breaker | page 중복·누락 | 동순위 key 없음 |
| INSERT | INSERT·RETURNING | key·constraint·parameter | returned ID·defaults | duplicate·FK 실패 |
| 1 row UPDATE | target SELECT·UPDATE | expected=1·current state | RETURNING·actual=1 | 0 또는 2+ |
| 삭제 | target SELECT·DELETE | 보존 정책·expected count | returned IDs·audit | WHERE 없음·count 이탈 |
| 여러 변경 | BEGIN·SAVEPOINT | invariant·lock·timeout | COMMIT·ROLLBACK·final SELECT | 중간 검증 실패 |

# 16. 한 장 요약

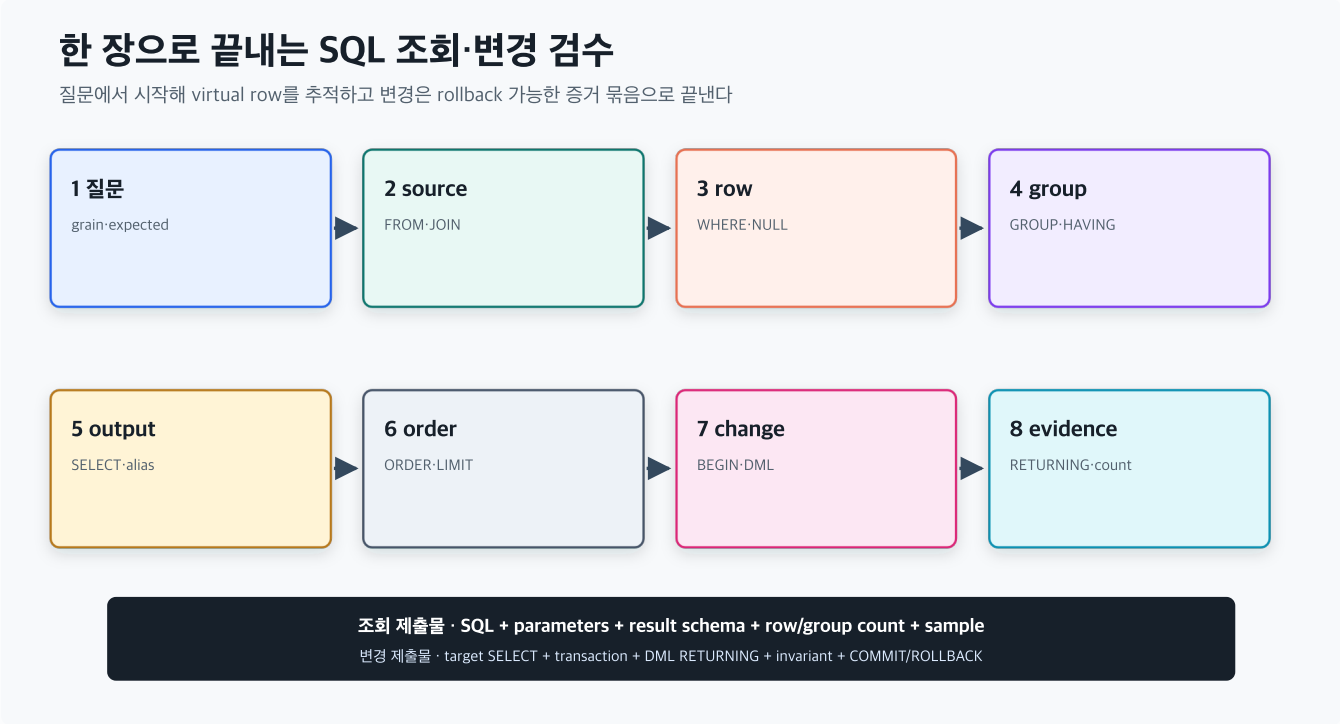
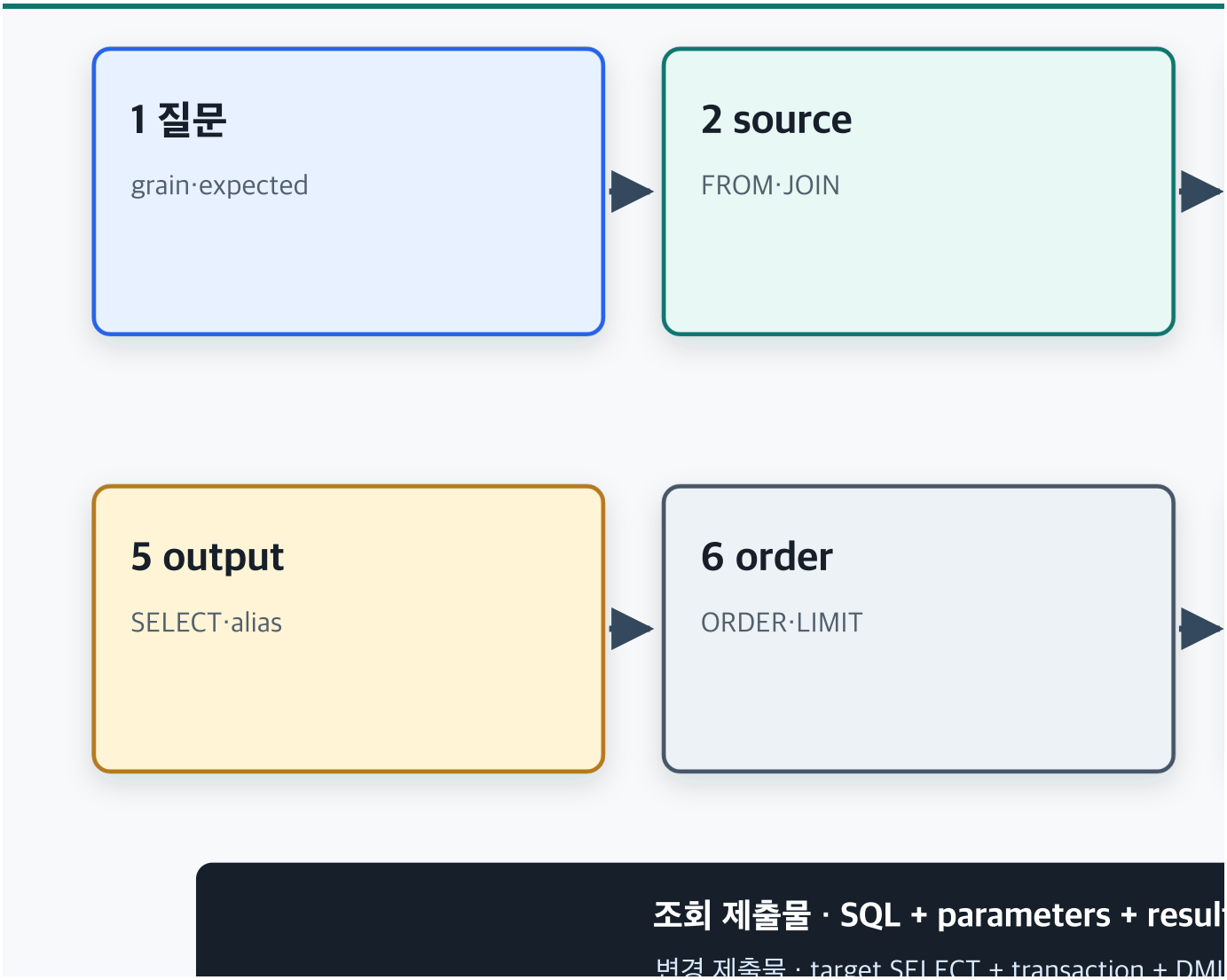
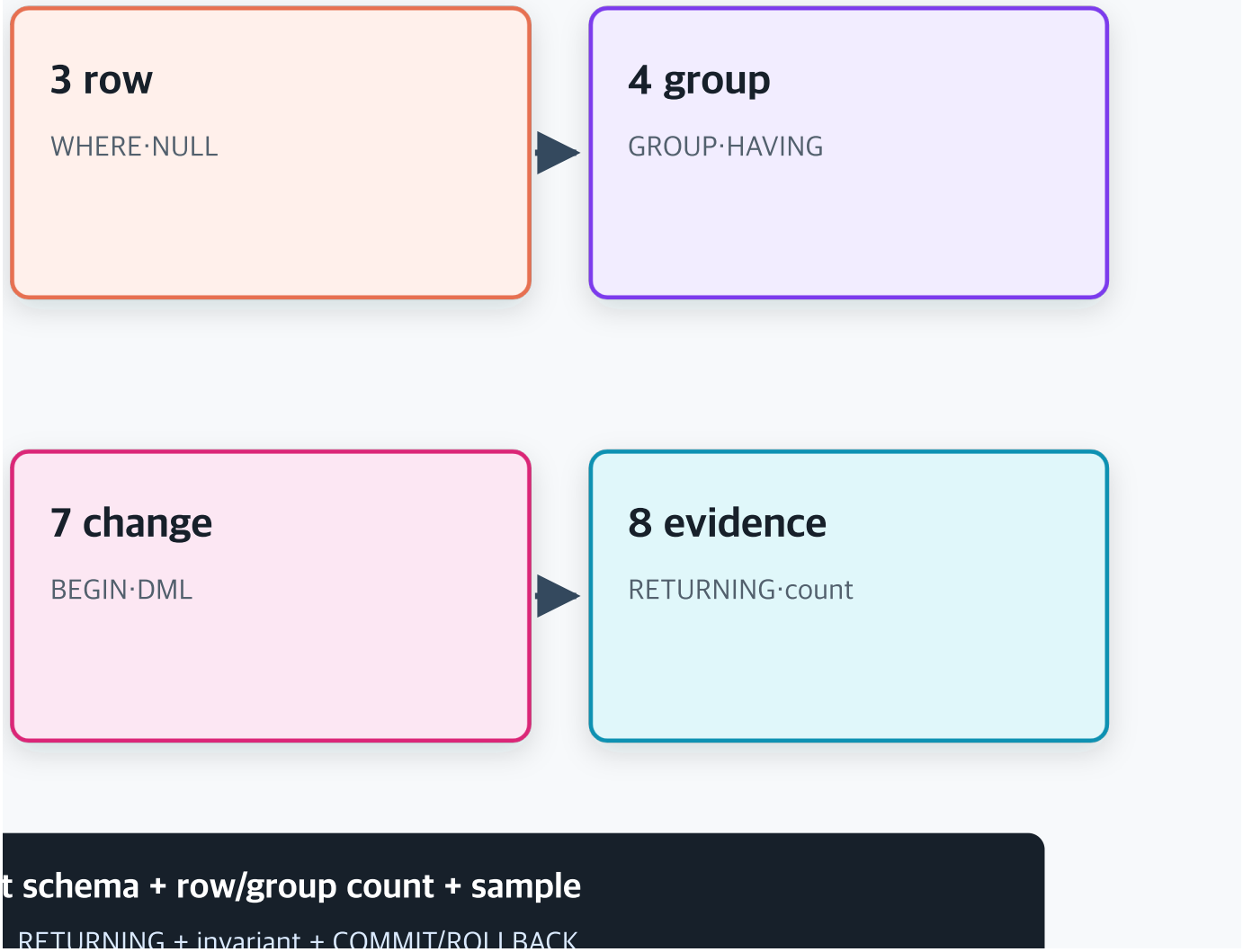



그림 15. SQL을 쓰기 전에 질문·grain·expected를 적고, 조회는 result schema·count·sample, 변경은 target SELECT·transaction·RETURNING·outcome으로 끝냅니다.


1 질문
grain·expected
2 source
FROM·JOIN
5 output
SELECT·alias
6 order
ORDER·LIMIT
조회 제출물 · SQL + parameters + resul
변경 제출물 · target SELECT + transaction + DML


3 row
WHERE·NULL
4 group
GROUP·HAVING
7 change
BEGIN·DML
8 evidence
RETURNING·count
t schema + row/group count + sample
RETURNING + invariant + COMMIT/ROLLBACK

## 조회 요약 문장

```
질문         : 무엇을 알고 싶은가
result grain  : 한 row·group은 무엇인가
source        : 어느 table·relation인가
row           : WHERE·NULL이 무엇을 남기나
group         : 집계 전·후 grain이 무엇인가
output        : column·alias·type은 무엇인가
order         : 동률까지 안정적인가
evidence      : SQL·params·schema·count·sample이 남았나
```

## 변경 요약 문장

```
target SELECT : 같은 WHERE의 ID·상태·count를 확인했나
transaction   : BEGIN과 확정 경계가 보이나
DML           : column·parameter·scope·precondition이 보이나
RETURNING     : 실제 변경 row를 받았나
invariant     : 변경 후 업무 규칙이 유지되나
decision      : expected=actual인가
outcome       : COMMIT 또는 ROLLBACK이 남았나
```

# 17. 셀프 테스트 · 12문제

## 문제 1 · 논리 순서

다음 clause를 result row가 만들어지는 논리적 이해 순서로 배치하세요.

```
SELECT · WHERE · ORDER BY · FROM/JOIN · GROUP BY/HAVING
```

답: _____

## 문제 2 · NULL

`WHERE completion_percent = NULL`이 NULL row를 찾지 못하는 이유와 올바른 문법을 쓰세요.

답: _____

## 문제 3 · AND·OR

상태가 active이면서 진도가 NULL 또는 0인 row만 찾는 WHERE를 쓰세요.

```
WHERE _____
```

## 문제 4 · LEFT JOIN

LEFT JOIN의 right table 상태 조건을 WHERE에 두었을 때 unmatched row가 사라질 수 있는 이유를 쓰세요.

답: _____

## 문제 5 · row 증폭

member 3 rows와 enrollment 4 rows를 INNER JOIN했을 때 결과가 4 rows인 이유를 grain으로 설명하세요.

답: _____

## 문제 6 · WHERE·HAVING

취소 row를 제외한 뒤 신청이 2건 이상인 강좌만 남길 때 취소 조건과 count 조건을 각각 어느 clause에 두어야 합니까?

답: _____

## 문제 7 · aggregate NULL

input이 `100, 80, NULL, 0`일 때 `COUNT(*)`, `COUNT(column)`, `AVG(column)`은 각각 몇입니까?

답: _____

## 문제 8 · stable order

`ORDER BY created_at DESC LIMIT 20`에서 created_at 동률을 안정적으로 해소하는 방법을 쓰세요.

답: _____

## 문제 9 · parameterization

값은 parameter로 보낼 수 있지만 table·column·sort direction은 일반적으로 그렇게 할 수 없습니다. 사용자 선택을 어떻게 처리합니까?

답: _____

## 문제 10 · safe UPDATE

expected 1 row UPDATE의 actual count가 0일 때 가능한 원인 두 가지와 다음 행동을 쓰세요.

답: _____

## 문제 11 · transaction

COMMIT과 ROLLBACK의 차이, SAVEPOINT의 역할을 각각 한 문장으로 쓰세요.

답: _____

## 문제 12 · dialect

실습기에서 SQL이 실행되었어도 운영 PostgreSQL에서 다시 검증해야 하는 요소를 네 가지 쓰세요.

답: _____

# 18. 셀프 테스트 정답·해설

## 정답 1

`FROM/JOIN → WHERE → GROUP BY/HAVING → SELECT → ORDER BY` 입니다. 이것은 결과를 이해하기 위한 논리 순서이며 optimizer의 물리 plan 순서와 같다는 뜻은 아닙니다.

## 정답 2

ordinary comparison에 NULL이 섞이면 결과가 UNKNOWN이고 WHERE는 TRUE만 남기므로 row가 제거됩니다. `WHERE completion_percent IS NULL` 을 사용합니다.

## 정답 3

```
WHERE status = 'active'
  AND (completion_percent IS NULL OR completion_percent = 0)
```

## 정답 4

unmatched row의 right column은 NULL입니다. `WHERE right.status = 'active'` 는 NULL 비교를 UNKNOWN으로 만들어 unmatched row를 제거합니다. 보존이 의도라면 match 조건을 ON에 둡니다.

## 정답 5

결과 grain은 한 회원이 아니라 한 수강 신청입니다. mem_1에 2개, mem_2에 2개, mem_3에 0개가 match하여 4 rows가 됩니다.

## 정답 6

취소 상태 조건은 개별 input row를 거르므로 WHERE, `COUNT(*) >= 2` 는 aggregate group을 거르므로 HAVING에 둡니다.

## 정답 7

`COUNT(*) = 4` , `COUNT(column) = 3` , `AVG(column) = (100+80+0)/3 = 60` 입니다. AVG는 NULL을 제외한 분모를 사용합니다.

### 정답 8

유일하고 안정적인 key를 tie-breaker로 추가합니다. 예: `ORDER BY created_at DESC, enrollment_id DESC LIMIT 20`.

### 정답 9

허용한 사용자 선택을 code allow-list의 고정 SQL fragment에 mapping합니다. 입력 문자열을 그대로 table·column·ORDER BY에 연결하지 않습니다.

### 정답 10

row가 없거나, tenant가 다르거나, 다른 요청이 current status를 먼저 바꾼 stale state일 수 있습니다. 자동 commit하지 않고 rollback한 뒤 현재 row와 사용자 의도를 재확인합니다.

### 정답 11

COMMIT은 transaction의 변경을 확정하고 ROLLBACK은 확정 전 변경을 취소합니다. SAVEPOINT는 transaction 전체가 아닌 특정 지점 이후만 되돌릴 수 있게 합니다.

### 정답 12

예: parameter placeholder, type·NULL behavior, DML·RETURNING 문법, JOIN·UPDATE·DELETE dialect, index·query plan, transaction isolation·concurrency, role·row security, 운영 schema·version입니다. 이 가운데 네 가지를 설명하면 됩니다.

## 19. 완료 체크리스트

### 조회

- [ ] 업무 질문을 한 문장으로 썼다.
- [ ] result grain을 `한 row = ...` 문장으로 썼다.
- [ ] input row·group·output row 수를 실행 전에 예상했다.
- [ ] SELECT column과 alias가 계약에 맞다.
- [ ] NULL·AND·OR·NOT IN의 UNKNOWN을 검토했다.
- [ ] JOIN cardinality와 unmatched row 보존을 설명했다.
- [ ] GROUP BY 전·후 grain과 NULL 분모를 정의했다.
- [ ] ORDER BY에 tie-breaker가 있다.

- [ ] SQL template와 parameters가 분리되었다.
- [ ] result schema·count·sample·NULL·duplicate 증거를 남겼다.

### 변경

- [ ] 같은 WHERE의 target SELECT를 실행했다.
- [ ] target PK·상태·tenant·expected count를 저장했다.
- [ ] INSERT의 column을 명시했다.
- [ ] UPDATE·DELETE에 WHERE·scope·current state precondition이 있다.
- [ ] parameterized query를 사용했다.
- [ ] BEGIN·COMMIT·ROLLBACK 경계를 확인했다.
- [ ] RETURNING 또는 후속 SELECT로 actual row를 확인했다.
- [ ] expected·actual count가 다르면 자동 commit하지 않는다.
- [ ] 변경 후 업무 불변식·constraint·audit를 검증했다.
- [ ] 최종 outcome을 COMMIT 또는 ROLLBACK으로 남겼다.

## 20. 공식 참고 자료

### PostgreSQL 18

- PostgreSQL Tutorial: The SQL Language
- PostgreSQL Queries
- Overview of query processing
- Table expressions: FROM, WHERE, GROUP BY, HAVING
- SELECT
- Sorting Rows: ORDER BY
- LIMIT and OFFSET
- Comparison Functions and Operators
- Logical Operators
- Data Manipulation
- INSERT
- UPDATE
- DELETE

- Returning Data from Modified Rows
- Transactions

## 보안·실습 엔진

- OWASP SQL Injection Prevention Cheat Sheet
- sql.js Documentation
- sql.js official repository

버전 기준일: 2026-07-15. 실습기는 sql.js 1.14.1의 단일 파일 asm.js build를 로컬에 고정하고 SHA-256을 검증했습니다. SQL 학습 결과는 실제 SQLite 3.49.1 memory database가 생성하며, 운영 구현은 선택한 PostgreSQL 버전·driver·framework·schema·권한·concurrency 환경에서 재검증합니다.