ScopeTreasure Courses

SQL COURSES · LESSON DB.5 · CODE SCOPE SERIES

JOINs — Do Tables Ko Jorna

Answer questions that live in two tables: INNER JOIN for matching rows only, LEFT JOIN for everyone — and the NULL that tells the truth.

Introduction

One table, one topic — that was the DB.1 rule. So students live in `students`, and their fee records live in `fees`, where each fee row carries a `student_id` pointing back at its student. Clean storage — but real questions cross the border: *show each student with his fee status*. The answer needs names from one table and fee facts from the other, side by side. Reconnecting tables at question time is the **JOIN** — the most useful idea in all of SQL.

Our `fees` table (from DB.2's shape) now holds September's records: Ali, Sara and Bilal have paid; Usman's fee is pending; **Hira has no fee row at all**. That one missing row is about to teach the difference between the two joins better than any diagram.

(Roman Urdu: JOIN = do tables ko sawal ke waqt jorna. INNER sirf milne wale, LEFT sab ko rakhta hai.)

Key points

- **A JOIN matches rows by a shared value:** the `ON` clause names the match — `ON students.id = fees.student_id`. The student whose `id` is 1 joins with every fee row whose `student_id` is 1.
- **INNER JOIN keeps only matches:** a student with no fee row disappears from the answer; a fee row pointing at no student would disappear too. INNER = the overlap.
- **LEFT JOIN keeps every row of the left table:** matched rows join normally; unmatched left rows still appear — with **NULL** (empty) in the right table's columns. LEFT = everyone on the left, partners where they exist.
  (Roman Urdu: LEFT JOIN mein left table ka har banda aayega; saathi mila to theek, warna NULL.)
- **NULL means "no value here":** it is not zero and not empty text — it is the absence of a row. In a LEFT JOIN result, NULL is information: *this student has no fee record yet.*
- **Table aliases shorten the writing:** `FROM students s JOIN fees f` lets you write `s.name` and `f.amount`. With two tables, prefix columns with their table (or alias) whenever a column name could belong to either.
- **JOIN + WHERE + ORDER BY compose:** a join builds the wide answer table; the WHERE, ORDER BY and LIMIT you already know then filter and arrange it. Everything from DB.4 still works — on the joined result.

Code examples

The September `fees` data used in every example below:

```
+----+------------+--------+--------+
| id | student_id | amount | status |
+----+------------+--------+--------+
|  1 |          1 |   5000 | Paid   |
|  2 |          2 |   5000 | Paid   |
|  3 |          3 |   4500 | Paid   |
|  4 |          5 |   5000 | Pending|
+----+------------+--------+--------+
```

Example 1 — INNER JOIN: only students with a fee record

```
SELECT students.name, fees.amount, fees.status
FROM students
INNER JOIN fees ON students.id = fees.student_id;
```

What it does: Each student is matched with fee rows carrying his id. Four fee rows exist, and all four point at real students — so the answer has four rows. Hira (id 4) has no fee row, so she is simply absent.

What you will see:

```
+-------------+--------+---------+
| name        | amount | status  |
+-------------+--------+---------+
| Ali Raza    |   5000 | Paid    |
| Sara Khan   |   5000 | Paid    |
| Bilal Ahmed |   4500 | Paid    |
| Usman Tariq |   5000 | Pending |
+-------------+--------+---------+
4 rows in set
```

Four rows — and the *missing* fifth row is the story. If the office prints this list as "students and fees", Hira silently vanishes. For money questions, silent vanishing is dangerous; that is exactly what the LEFT JOIN fixes.

Example 2 — LEFT JOIN: every student, fee or no fee

```
SELECT students.name, fees.amount, fees.status
FROM students
LEFT JOIN fees ON students.id = fees.student_id;
```

What it does: Same match, one word changed. Now all five students appear. The four with fee rows look identical to Example 1; Hira appears with NULL in both fee columns — the database's honest way of saying "no record exists".

What you will see:

```
+-------------+--------+---------+
| name        | amount | status  |
+-------------+--------+---------+
| Ali Raza    |   5000 | Paid    |
| Sara Khan   |   5000 | Paid    |
| Bilal Ahmed |   4500 | Paid    |
| Hira Shah   |   NULL | NULL    |
| Usman Tariq |   5000 | Pending |
+-------------+--------+---------+
5 rows in set
```

Five rows, and the NULL row is the most valuable one on the page: it is the office's reminder list. Any report about *people* (not just payments) should start from LEFT JOIN thinking.

Example 3 — JOIN with WHERE and ORDER BY: the paid list, biggest first

```
SELECT s.name, f.amount
FROM students s
INNER JOIN fees f ON s.id = f.student_id
WHERE f.status = 'Paid'
ORDER BY f.amount DESC;
```

What it does: Aliases (`s`, `f`) keep the query short. The join builds the wide table, WHERE keeps only the Paid rows, and ORDER BY arranges them by amount. Ali and Sara tie at 5000 — tied rows can appear in either order between themselves.

What you will see:

```
+-------------+--------+
| name        | amount |
+-------------+--------+
| Ali Raza    |   5000 |
| Sara Khan   |   5000 |
| Bilal Ahmed |   4500 |
+-------------+--------+
3 rows in set
```

Usman's Pending row is filtered out by the WHERE; Hira never even entered (INNER JOIN). Same tables, three different questions, three different honest answers — the join type is part of the question.

Quick recap

- **JOIN** reconnects tables at question time via `ON left.key = right.key`.
- **INNER JOIN** = matches only; unmatched rows on both sides vanish.

- **LEFT JOIN** = every left row; missing partners show **NULL**.
- Aliases (`s`, `f`) and DB.4's WHERE / ORDER BY / LIMIT all work on the joined result.

Common mistakes

- **Forgetting the ON clause:** a join without a match condition pairs *every* row with *every* row (5 students × 4 fees = 20 nonsense rows). This is called a cross join, and it is almost never the question you meant to ask.
- **Using INNER JOIN for a people report:** students without records silently disappear — the report looks complete and is not. If the list is about people, think LEFT JOIN first.
  (Roman Urdu: INNER JOIN mein bina record wala banda chup chaap ghayab — report adhoori, pata bhi nahi chalta.)
- **Reading NULL as zero:** Hira's amount is not 0 — she has *no fee row*. Averages and sums treat these differently; so should you.
- **Ambiguous column names:** both tables have an `id`. Writing just `id` in a join query fails with an "ambiguous" error — prefix it: `students.id` or `s.id`.

Practice

1. Write the LEFT JOIN query that lists every student with his fee status, then add a WHERE that keeps only the rows where the fee side is NULL (`WHERE f.student_id IS NULL`). Who is on the reminder list?
2. Write an INNER JOIN query showing student name and fee amount for students from Sahiwal only. (Predict first: which of the Sahiwal students actually has a fee row?)
3. Explain in two lines: a query returns 4 rows with INNER JOIN and 5 with LEFT JOIN on the same tables. What exactly is the fifth row, and what do its NULLs mean?

Practice quiz

1. Show students together with their fee rows (only students who have fees).

- SELECT * FROM students INNER JOIN fees ON students.id = fees.student_id;
- SELECT * FROM students LEFT JOIN fees;
- SELECT * FROM students, fees;
- SELECT * FROM students WHERE fees;

Answer: SELECT * FROM students INNER JOIN fees ON students.id = fees.student_id;

Watch the Code Scope channel: https://www.youtube.com/@CodeScopePK — this lesson's own video joins the channel when it is recorded.

Read online: https://scopetreasure.com/notes/sql-joins · More free tools and notes at https://scopetreasure.com