SQL JOINs

The Theory Behind Combining Tables

What Problem Does JOIN Solve?

Relational databases store data in **separate tables** to avoid redundancy.

- User names → `Users` table
- Loan records → `Loans` table
- Book details → `Books` table

To answer questions like *"Which users borrowed which books?"*, we need to **combine** rows from multiple tables.

→ That is exactly what **JOIN** does.

The Core Idea

A JOIN links rows from two tables by **matching values** in related columns.

Users

id	name
1	Alice
2	Bob
3	Carol

Loans

id	user_id	book_id
1	1	101
2	1	102
3	2	103

`Users.id` and `Loans.user_id` are the **matching columns** (the join key).

What is a Join Key?

- A **join key** is the column (or set of columns) used to match rows between two tables
- Usually a **primary key** in one table and a **foreign key** in the other
- Rows with **equal key values** are considered related

```
Users.id   ←→   Loans.user_id
Books.id   ←→   Loans.book_id
```

The Result of a JOIN

Combining rows creates a **wider** result: columns from both tables side by side.

name	book_id	loan_date
Alice	101	2024-01-10
Alice	102	2024-01-15
Bob	103	2024-02-01

Carol does not appear – she has no loans (more on this soon).

Six Types of JOIN

Each type answers a different question about which rows to include:

Type	Question answered
`INNER JOIN`	Which rows match in **both** tables?
`LEFT JOIN`	All rows from the **left** table, matches or not
`RIGHT JOIN`	All rows from the **right** table, matches or not
`FULL JOIN`	All rows from **either** table, matches or not
`CROSS JOIN`	Every possible **combination** of rows
`SELF JOIN`	Rows from a table matched **against itself**

INNER JOIN

The Intersection

INNER JOIN - Concept

Returns only rows where a **match exists in both tables**.

Rows with no match in the other table are **excluded**.

Left table

A	key
1	X
2	Y
3	Z
	W

← no match, excluded

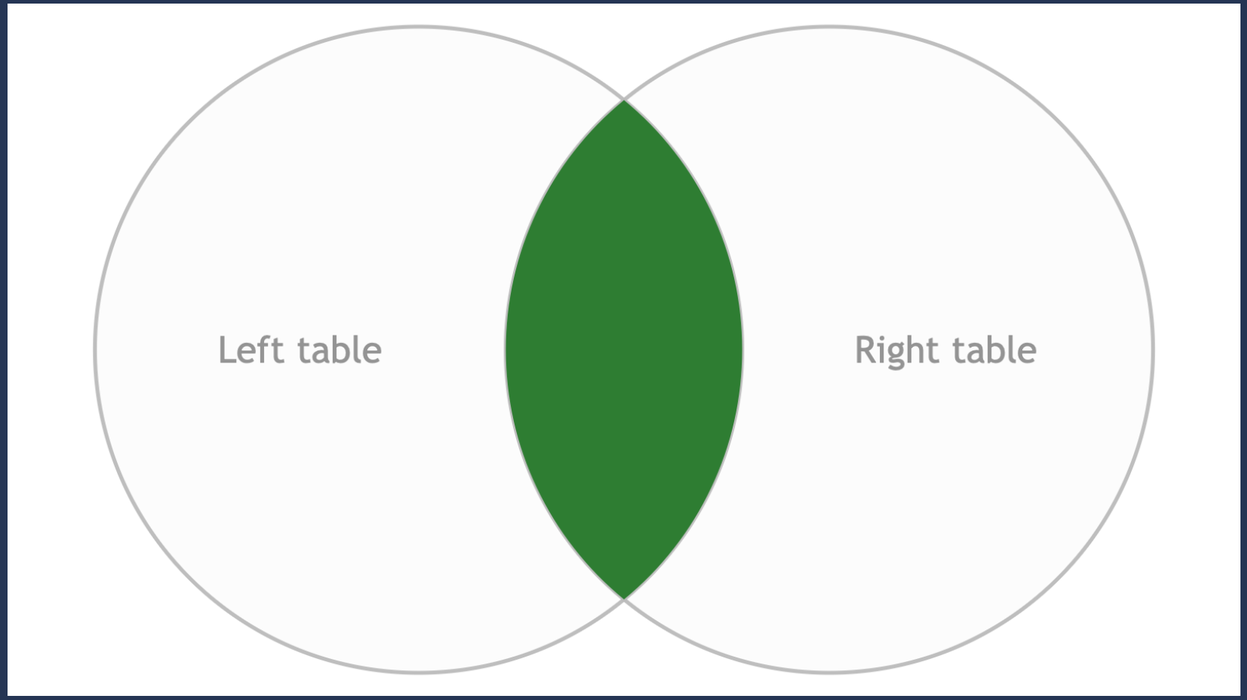
Right table

key	B
X	
Z	

Result

A	key	B
1	X	
2	Y	
3	Z	

INNER JOIN - Visualised



Think of it as the **intersection** of two sets

INNER JOIN - Real Example

"Show the names of users who have made at least one loan, together with the loan date."

Users

id	name
1	Alice
2	Bob
3	Carol

Loans

id	user_id	loan_date
1	1	2024-01-10
2	2	2024-02-01

Result (INNER JOIN on `Users.id = Loans.user_id`):

name	loan_date
Alice	2024-01-10
Bob	2024-02-01

Carol has no loans → she is **not in the result**.

INNER JOIN – When to Use It

- When you only care about rows that have a **corresponding record** in the related table
- When missing matches should be **silently ignored**
- Most common JOIN in everyday queries

⚠️ If a row has no match, it **disappears from the result** – be aware of this silent exclusion.

LEFT JOIN

Keep Everything on the Left

LEFT JOIN - Concept

Returns **all rows from the left table**, regardless of whether they have a match in the right table.

For rows with no match: the right-side columns are filled with **NULL**.

Left table

A	key
1	X
2	Y
3	Z

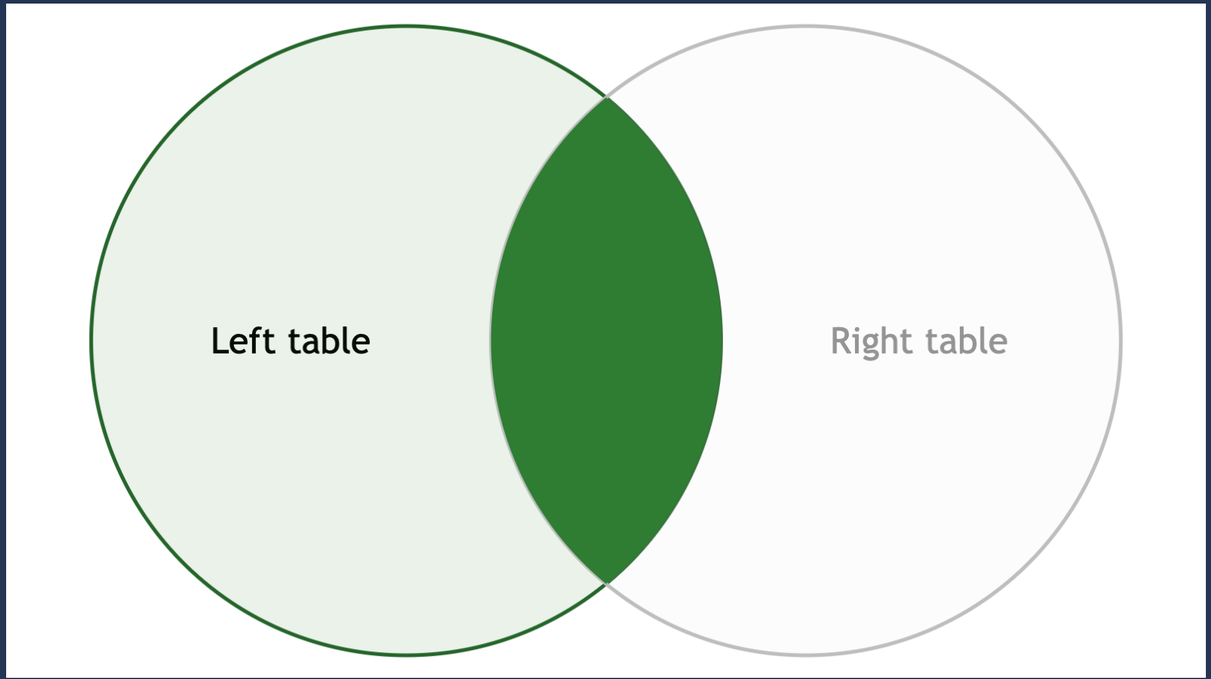
Right table

key	B
X	🔵
Z	🟢

Result

A	key	B
1	X	🔵
2	Y	NULL ← no match
3	Z	🟢

LEFT JOIN - Visualised



📌 The left table is always **complete** in the result

LEFT JOIN - Real Example

"Show all books, including those that have never been borrowed."

Books

id	title
101	1984
102	Dune
103	Neuromancer

Loans

id	book_id
1	101
2	101

Result (LEFT JOIN on `Books.id = Loans.book_id`):

title	loan_id	
1984	1	
1984	2	
Dune	NULL	← never borrowed
Neuromancer	NULL	← never borrowed

LEFT JOIN – Finding Missing Data

A powerful pattern: **filter for NULL** to find rows with no match.

"Show books that have *never** been borrowed."*

```
SELECT b.title
FROM Books b
LEFT JOIN Loans l ON b.id = l.book_id
WHERE l.id IS NULL;
```

NULL in the right-side column means **no matching row was found**.
This is one of the most common uses of LEFT JOIN.

LEFT JOIN – When to Use It

- When you want to keep **all records** from the main table
- When missing related data is normal (e.g. books not yet borrowed)
- When you want to **detect missing links** (filter WHERE right.id IS NULL)

⚠️ A book borrowed 5 times will appear **5 times** in the result – one row per loan.

RIGHT JOIN

Keep Everything on the Right

RIGHT JOIN - Concept

The **mirror** of LEFT JOIN.

Returns **all rows from the right table**, with NULLs for unmatched left-side columns.

Left table

A	key
1	X
3	Z

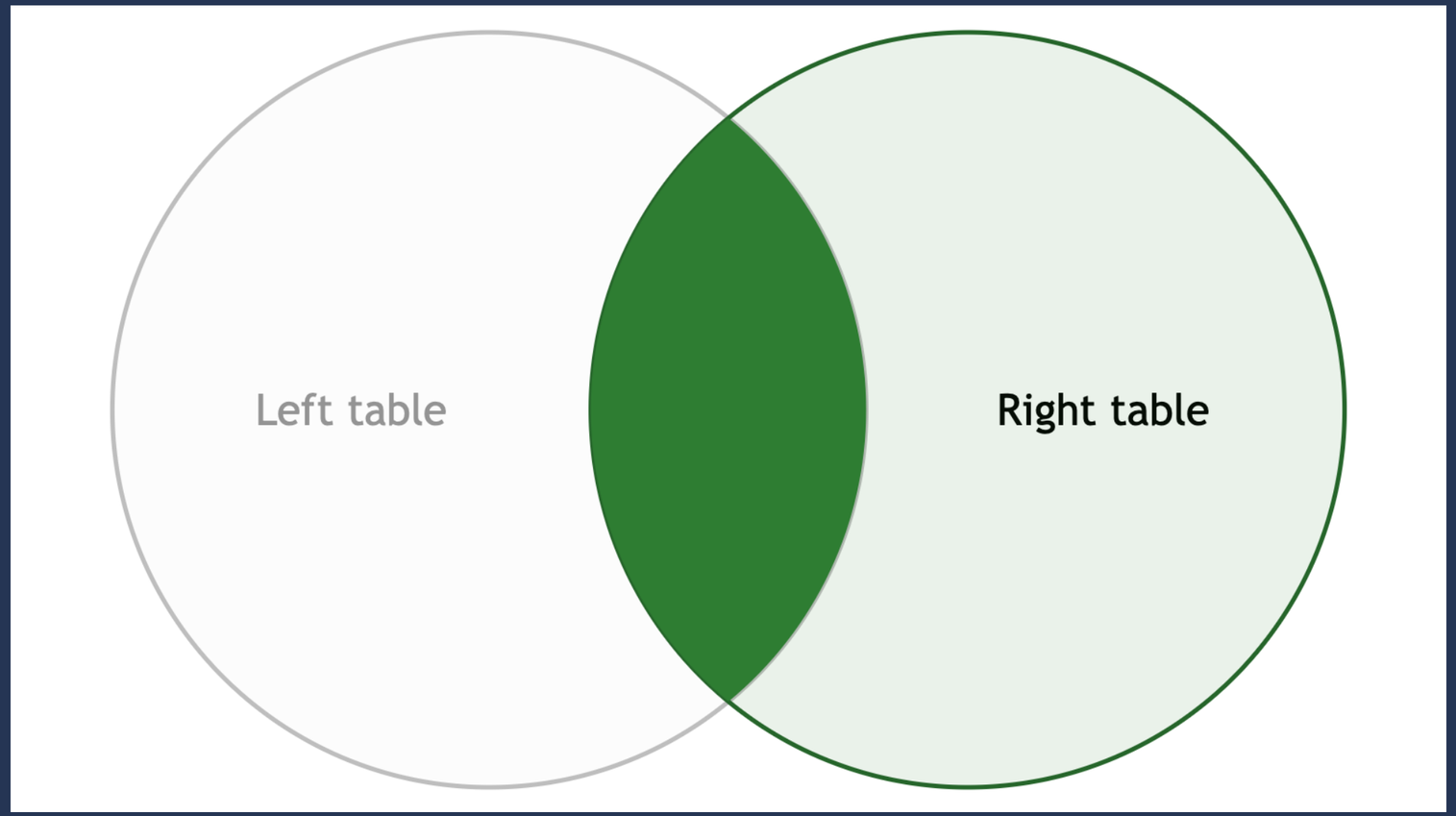
Right table

key	B
X	🔵
Y	🟠
Z	🟢

Result

A	key	B	
1	X	🔵	
NULL	Y	🟠	← no left match
3	Z	🟢	

RIGHT JOIN - Visualised



RIGHT JOIN - Practical Note

`A RIGHT JOIN B` is always equivalent to `B LEFT JOIN A`

Most developers prefer to **rewrite RIGHT JOINs as LEFT JOINs** by swapping the table order – the logic is easier to read left-to-right.

```
Books RIGHT JOIN Loans
   ≡
Loans LEFT JOIN Books
```

RIGHT JOIN – When to Use It

- When you want to keep **all records** from the second (right) table
- Rare in practice – most queries are rewritten using LEFT JOIN instead

In **SQLite** versions before 3.39.0, RIGHT JOIN is not supported.
→ Always rewrite as `right_table LEFT JOIN left_table`.

FULL JOIN

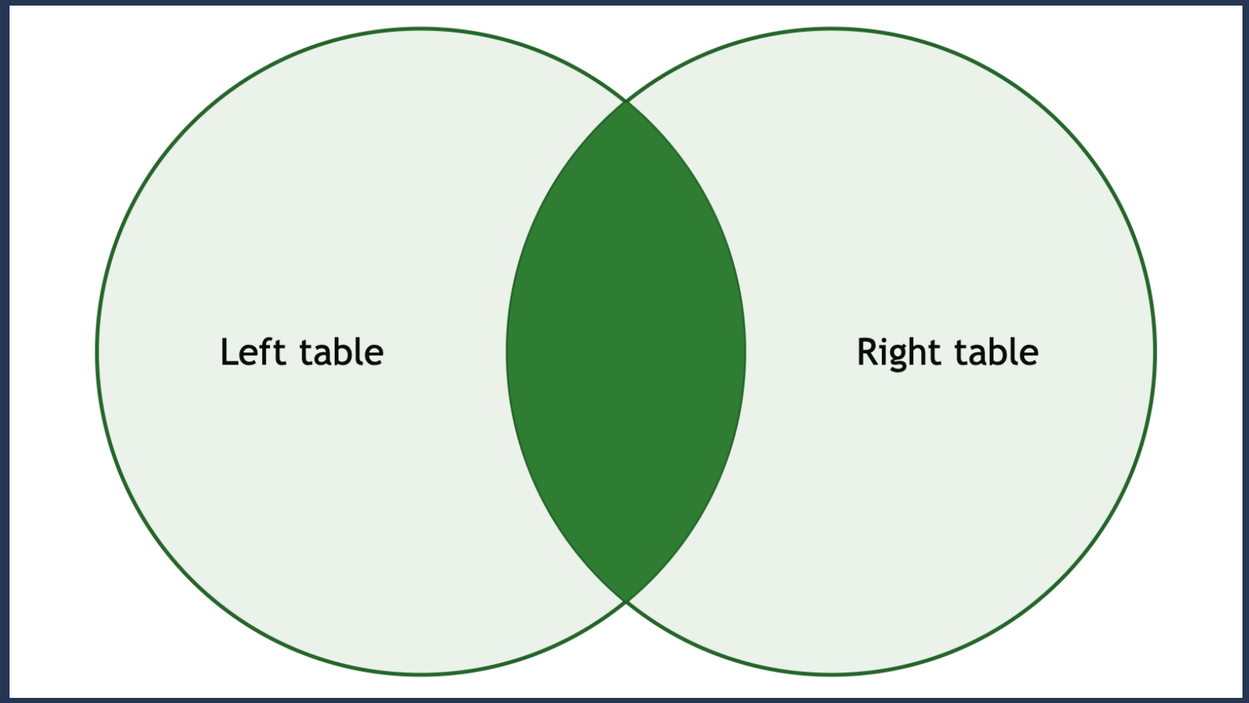
Keep Everything from Both Sides

FULL JOIN - Concept

Returns **all rows from both tables**.

- Matching rows → columns from both tables are populated
- Left-only rows → right columns are NULL
- Right-only rows → left columns are NULL

FULL JOIN - Visualised



📌 Think of it as the **union** of LEFT JOIN and RIGHT JOIN

FULL JOIN - Real Example

"Show all books and all loans — even books never borrowed and loans referencing deleted books."

Books

id	title
101	1984
102	Dune

Loans

id	book_id	
1	101	
2	999	← orphan (book 999 deleted)

Result (FULL JOIN):

title	loan_id	book_id	
1984	1	101	
Dune	NULL	NULL	← never borrowed
NULL	2	999	← orphan loan

FULL JOIN - When to Use It

- **Data auditing** – find unmatched rows on either side
- **Reconciliation** – comparing two datasets for gaps
- Less common in everyday application queries

⚠️ Results can be confusing – NULL appears in columns from both sides.
⚠️ Not supported in SQLite < 3.39.0 (emulate with UNION of two LEFT JOINs).

CROSS JOIN

The Cartesian Product

CROSS JOIN - Concept

Combines **every row** from the left table with **every row** from the right table.

No matching condition – every possible pair is generated.

Left	Right	Result (all combinations)
A	X	A – X
B	Y	A – Y
	Z	B – X
		B – Y

If Left has **m** rows and Right has **n** rows → result has **m × n** rows.

CROSS JOIN – When to Use It

- **Generating all combinations**: e.g. every session paired with every lane
- **Seeding test data**: build a full matrix of values
- **Schedule generation**: every time slot × every resource

⚠️ Can produce **very large results** – a 1000-row table CROSS JOIN a 1000-row table gives 1,000,000 rows.
⚠️ Almost never used without additional filtering.

INNER JOIN vs CROSS JOIN

Both combine rows from two tables – but in fundamentally different ways.

	INNER JOIN	CROSS JOIN
Match condition	Yes – rows must share a key value	No – all pairs are generated
Row count	≤ rows in either table	Exactly m × n rows
Typical use	Retrieve related data	Generate all combinations
Risk	Silent row loss (no match)	Combinatorial explosion

- INNER JOIN is a **filtered** CROSS JOIN – it takes all pairs and keeps only those where the condition holds.
- A CROSS JOIN with a `WHERE` clause that matches keys is logically equivalent to an INNER JOIN.

SELF JOIN

A Table Joined With Itself

SELF JOIN - Concept

The same table appears **twice**, with different aliases, to compare rows within the same table.

Employees

id	name	manager_id
1	Alice	NULL
2	Bob	1
3	Carol	1
4	Dave	2

📌 `manager_id` points back to another row in the **same table**

SELF JOIN - Visualised

```
Employees (as "e")     Employees (as "m")
   [employee]      →       [their manager]
```

```
e.id | e.name      joined on      m.id | m.name
─────┼──────────   e.manager_id = m.id  ─────┼──────────
  2  | Bob                          1  | Alice
  3  | Carol                        1  | Alice
  4  | Dave                         2  | Bob
```

SELF JOIN – When to Use It

- **Hierarchical data**: employees and their managers, categories and subcategories
- **Finding pairs**: users with the same role, products in the same category
- **Comparing adjacent rows**: sequences, rankings

Always use table aliases (e.g. `Employees e`, `Employees m`) – without aliases the query is ambiguous.

What is an Outer JOIN?

LEFT JOIN, RIGHT JOIN, and FULL JOIN are collectively called **outer joins**.

The term *outer* refers to the rows that have **no matching partner** in the other table – the "outsiders". In an inner join these rows are discarded; in an outer join they are **kept**, with NULLs filling the missing side.

Category	JOIN types	Unmatched rows
Inner join	`INNER JOIN`	Discarded
Outer joins	`LEFT`, `RIGHT`, `FULL JOIN`	Kept (with NULLs)
Other	`CROSS`, `SELF`	No match condition

NULL in Outer JOINs

A key concept to understand when working with LEFT, RIGHT, and FULL JOINs.

📌 When no match is found, the **missing side's columns are filled with NULL**

📌 NULL means *"there is no related row"* — not zero, not empty string

📌 To detect unmatched rows, use `IS NULL` (not `= NULL`)

```
-- Find users with no loans
WHERE l.id IS NULL        ✅ correct
WHERE l.id = NULL         ❌ always false
```

JOIN vs WHERE: Two Ways to Filter

Both `JOIN ... ON` and `WHERE` can restrict rows – but they behave differently in outer joins.

JOIN condition – applied during the join itself (controls what gets NULL vs excluded)
WHERE condition – applied after the join (can eliminate NULLs you wanted to keep)

JOIN vs WHERE: Example

Goal: show all books, with their loan date only if the loan was in 2024.

```sql
-- ✅ Correct: filter in the ON clause
SELECT b.title, l.loan_date
FROM Books b
LEFT JOIN Loans l ON b.id = l.book_id
                  AND l.loan_date LIKE '2024%';
```

```sql
-- ❌ Wrong: filter in WHERE eliminates unmatched books
SELECT b.title, l.loan_date
FROM Books b
LEFT JOIN Loans l ON b.id = l.book_id
WHERE l.loan_date LIKE '2024%';
```

📌 In the second query, books with no 2024 loan are silently removed.

Joining More Than Two Tables

JOINs chain naturally — each new JOIN adds another table to the result.

```
Users → Loans → Books
```

```
Result columns:
  user name | loan date | book title | book genre
```

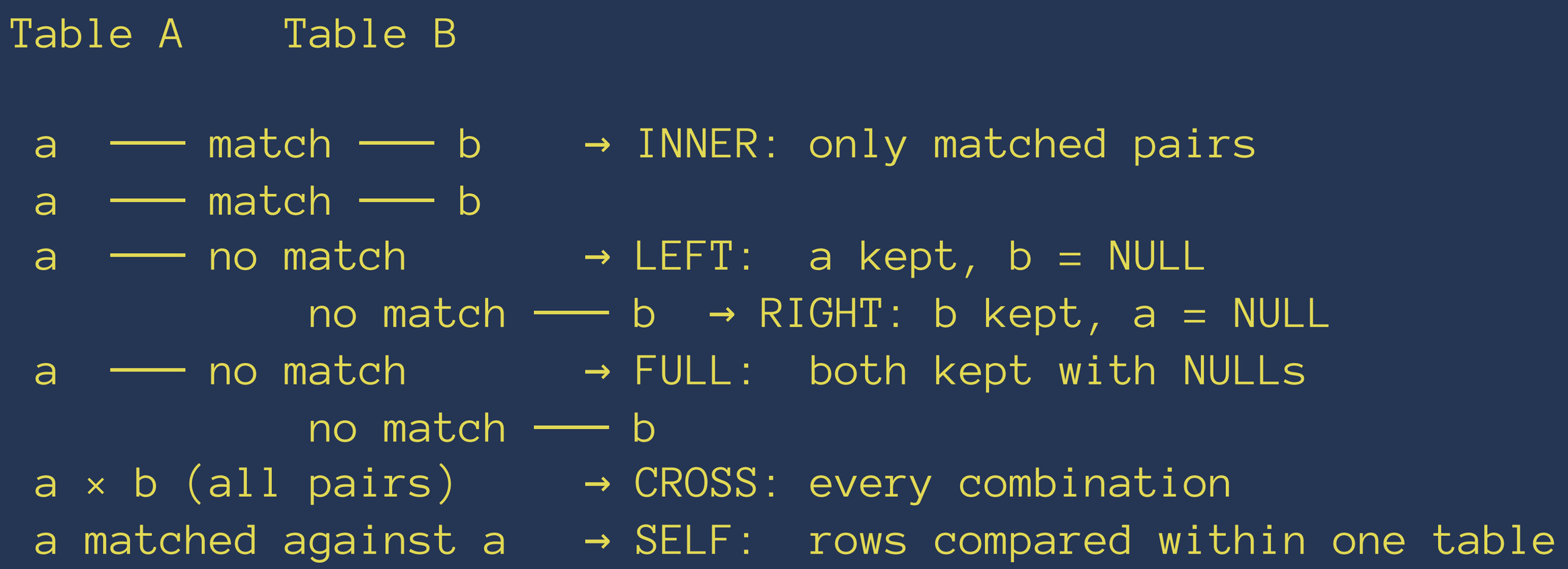
📌 Each JOIN adds one more relationship link
📌 The order matters for readability, not correctness
📌 Start from the most central table and join outward

How Many Rows Will the Result Have?

Understanding result size prevents surprises:

Scenario	Effect on row count
One-to-one match	Same number of rows
One-to-many match	**Multiplied** – one row per related record
No match (INNER)	Row is **removed**
No match (LEFT/RIGHT)	Row is **kept with NULLs**
CROSS JOIN	**m × n** rows

JOIN Types at a Glance



Choosing the Right JOIN

Question	JOIN to use
"Give me X with their related Y"	`INNER JOIN`
"Give me all X, with Y if it exists"	`LEFT JOIN`
"Which X have no Y?"	`LEFT JOIN` + `WHERE y.id IS NULL`
"Give me everything from both tables"	`FULL JOIN`
"Generate all possible pairs of X and Y"	`CROSS JOIN`
"Compare rows within the same table"	`SELF JOIN`

Summary

✅ JOIN combines rows from two tables by **matching on a common key**
✅ **INNER JOIN** – only matching rows (the intersection)
✅ **LEFT JOIN** – all rows from the left table, NULLs for missing right rows
✅ **RIGHT JOIN** – all rows from the right table, NULLs for missing left rows
✅ **FULL JOIN** – all rows from both tables (the full union)
✅ **CROSS JOIN** – every row × every row (Cartesian product)