

chapter five

The Mental Model of a Join

Part I operated on one table at a time. A join combines related rows from two tables into wider result rows. This chapter develops a physical model for that operation before introducing the syntax used to express it.

Why Your Data Lives in Pieces

Suppose the shop kept one row per item sold and copied the customer's name, country, and signup date onto every row, along with the product's name, category, and price.

Repeated customer facts can drift apart.

Ada's name and country would appear on three item rows. If she moved from the US to Canada, every copy would need updating; a missed row would leave conflicting values. The table would also have nowhere to store Margaret, who has signed up but placed no order.

everything.xlsx (one big table)

customer	country	product	price
Ada	US	Keyboard	89
Ada	US	Mouse	25
Ada	US	Desk Lamp	45
Grace	UK	Notebook	12
Linus	FI	Monitor	349

the same fact,
copied 3 times!

?



Margaret
signed up, never ordered:
no row for her at all

A flat sales table repeats customer facts and has no row for Margaret.

The normalized design stores each customer once in `customers` and each order once in `orders`. The tables are connected by a foreign key: each order's `customer_id` refers to one row in `customers`.

`orders.customer_id` identifies the related customer row.

Here are the two tables we'll use throughout this chapter (we'll show just the columns that matter for the join):

customers

customer_id	name	country
1	Ada	US
2	Grace	UK
3	Linus	FI
4	Margaret	US
5	Dennis	US

orders

order_id	customer_id	order_date	status
1001	1	2023-03-01	shipped
1002	1	2023-03-15	shipped
1003	2	2023-03-20	cancelled
1004	3	2023-04-02	shipped
1005	5	2023-04-10	pending

The answer to "who placed order 1003?" is now spread across two tables. The order row contains `customer_id` 2; the name Grace is stored in `customers`. A join combines those related rows in the query result.

remember this!

Normalized tables store related facts separately. A join combines matching rows in a query result without copying them into permanent storage.

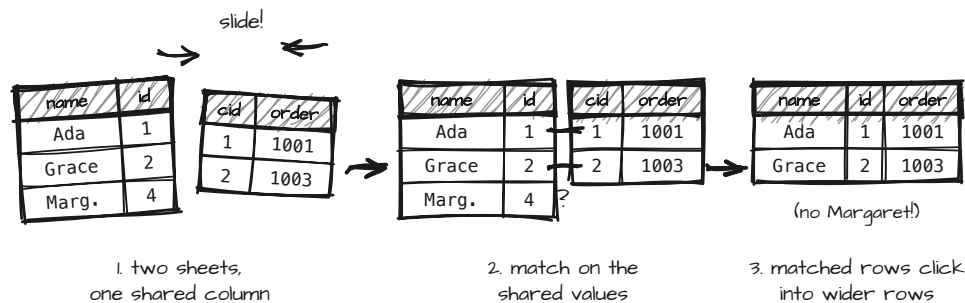
Two Grids on a Table

Picture `customers` and `orders` printed on separate sheets. Place `customers` on the left and `orders` on the right. Both sheets contain a `customer_id` column.

For each order, use its `customer_id` to find the customer row with the same value. Order 1001 has customer id 1, so it lines up with Ada. Order 1003 has id 2 and lines up with Grace. Each matching pair forms one wider result row with columns from both tables.

Match related values;
combine the two
rows.

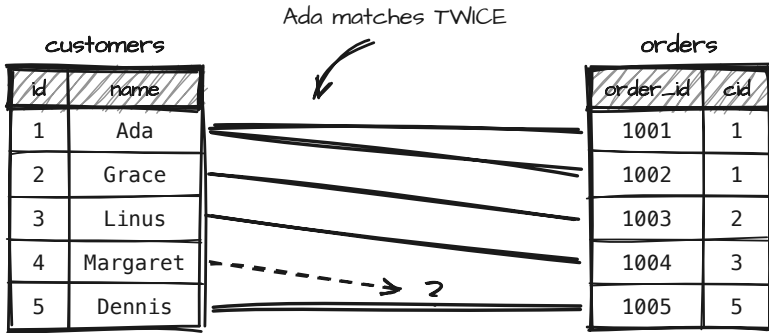
The sheets provide the chapter's working model: align two grids by related columns and combine the matching rows. Join syntax identifies the tables and states the matching condition.



Align the `customer_id` columns, match equal values, and combine each pair.

Join type decides
whether Margaret
remains.

Ada's row is used twice because two orders refer to her. One row on the left may therefore match several rows on the right. Margaret's row has no match because no order contains customer id 4. The join type determines whether an unmatched row such as hers remains in the result.



one left row can match many right rows -
or none at all

Ada matches two orders; Margaret has no matching order.

The Cartesian Product: Every Possible Pairing

A **Cartesian product** pairs every row from one table with every row from the other. SQL produces it directly with **CROSS JOIN**:

```
SELECT c.name, o.order_id, o.customer_id
FROM customers c
CROSS JOIN orders o;
```

Five customers crossed with five orders produce 25 rows. For a shorter example, use the first two customers and first three orders:

customers (first two rows)

customer_id	name
1	Ada
2	Grace

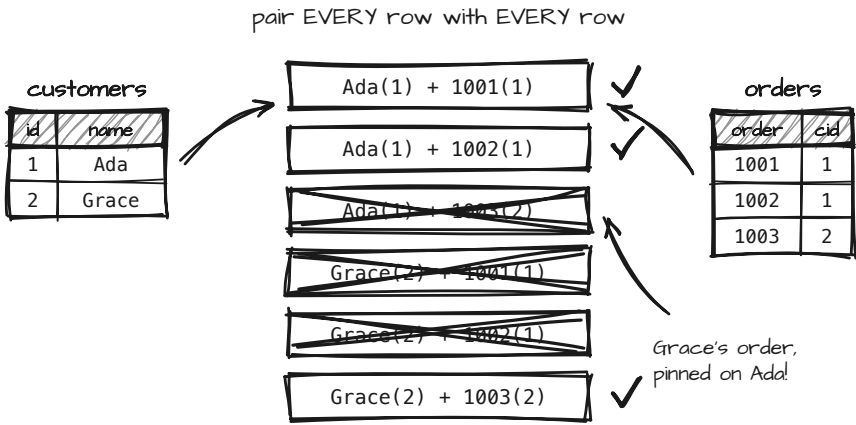
orders (first three rows)

order_id	customer_id	status
1001	1	shipped
1002	1	shipped
1003	2	cancelled

The cross join of these produces every combination — $2 \times 3 = 6$ rows:

Result of the cross join

name	order_id	customer_id
Ada	1001	1
Ada	1002	1
Ada	1003	2
Grace	1001	1
Grace	1002	1
Grace	1003	2



$2 \times 3 = 6$ pairings - only 3 agree on `customer_id`

A Cartesian product pairs every customer with every order; only three of the six pairs agree on `customer_id`.

Most pairs do not represent the actual relationship. "Ada, order 1003," for example, combines Ada with an order whose `customer_id` is 2.

Three pairs do match: Ada with 1001 and 1002, and Grace with 1003. In those rows, the customer id agrees on both sides.

Keep only pairs whose customer ids agree.

An inner equijoin can be understood logically as a Cartesian product filtered to the pairs whose key values agree:

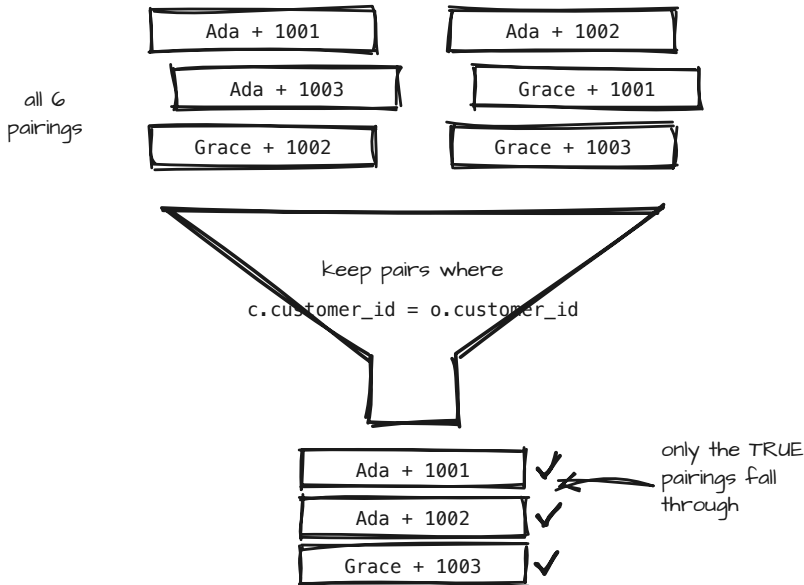
```
SELECT c.name, o.order_id
FROM customers c
CROSS JOIN orders o
WHERE c.customer_id = o.customer_id;
```

Result: only matching pairs remain

name	order_id
Ada	1001
Ada	1002
Grace	1003

A database normally uses a more efficient join algorithm rather than materializing every pair. The result is nevertheless the same as filtering the Cartesian product by this condition.

The Mental Model of a Join



The matching condition removes customer-order pairs whose ids differ.

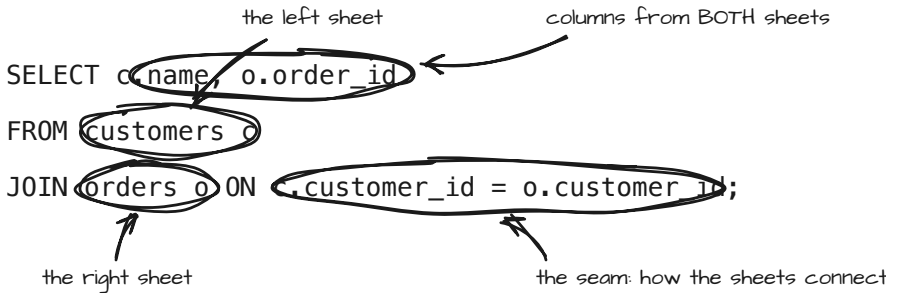
remember this!

An inner equijoin returns the matching pairs from the Cartesian product. The database may use a more efficient algorithm to produce the same result.

The ON Clause: Saying What Must Match

Writing the match in `WHERE` after a `CROSS JOIN` works, but modern join syntax puts the relationship next to the joined table in an `ON` clause.

```
SELECT c.name, o.order_id
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id;
```



The ON clause states how rows from the two tables match.

Read it as "from customers, join orders where the customer ids are equal." The `ON` clause states the relationship between the sheets.

It often compares a foreign key with the referenced key. An `ON` clause may also contain other conditions, as later chapters will show.

`ON` states the row-matching condition.

With the full tables, the query also pairs Linus with order 1004 and Dennis with order 1005. Here, `JOIN` is shorthand for `INNER JOIN`. The pattern is `FROM` one table, `JOIN` another, and `ON` the matching condition.

Aliases and Qualified Names

The queries above use table aliases and qualified column names.

Writing `FROM customers c` gives customers the alias `c` for the rest of the query. Short aliases usually echo the table name: `orders o`, `customers c`, and `products p`. The optional `AS` (`customers AS c`) is also valid. Use one style consistently.

Once two tables are in play, a bare column name may be ambiguous. Both tables have `customer_id`, so PostgreSQL rejects an unqualified reference. Prefix the column with

c.name reads as "the name belonging to c."

its table alias: `c.customer_id` or `o.customer_id`. The name `c.name` means the `name` column from the table aliased as `c`.

This book qualifies every column in multi-table queries, even when the bare name would be unambiguous. `o.status` makes its source explicit and remains clear if another table later gains a `status` column.

```
SELECT c.name, c.country, o.order_id, o.status
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id;
```

Result

name	country	order_id	status
Ada	US	1001	shipped
Ada	US	1002	shipped
Grace	UK	1003	cancelled
Linus	FI	1004	shipped
Dennis	US	1005	pending

Each result row combines an order with the matching customer's name and country. Margaret is absent because her customer row matches no order. Chapters 6 and 7 show how the join type controls that outcome.

remember this!

Alias every table (`customers c`) and qualify every column (`c.name`) in any query that touches more than one table. It prevents ambiguity errors, and it lets anyone reading the query trace each value back to its source sheet at a glance.

Where We're Headed

The central model is two grids aligned by related columns, producing one result row for each matching pair.

The next chapters vary which pairs and unmatched rows are retained:

- **Chapter 6** asks: what if we keep *only* the matched pairs? (INNER JOIN — what you just saw.)
- **Chapter 7** asks: what about rows like Margaret's, with no partner? (LEFT, RIGHT, and FULL OUTER joins.)
- **Chapter 8** slides a table against *itself*, and chains three or four sheets in a row.
- **Chapter 9** covers common mistakes and a checklist for finding them.

Exercises

Exercise 5.1. A `CROSS JOIN` of the full `customers` table (5 rows) and the full `orders` table (5 rows) produces how many rows? Of those, how many are "true" pairings where `c.customer_id = o.customer_id`, and what does that tell you about the raw material a join is carved from?

Exercise 5.2. Write a query using modern `JOIN ... ON` syntax that pairs each order with the customer who placed it, returning the customer's `name` and the `order_id`. Alias both tables.

Exercise 5.3. Both `customers` and `orders` contain a column called `customer_id`. Explain why a query joining them must qualify that column as `c.customer_id` or `o.customer_id`, and what error you get if you write it bare.

Exercise 5.4. Rewrite this legacy comma join as a `JOIN ... ON` statement, and explain what disaster occurs if the `WHERE` line is accidentally deleted from the comma version, then contrast that with what happens if the `ON` clause is deleted from the modern version.

```
SELECT c.name, o.order_id  
FROM customers c, orders o  
WHERE c.customer_id = o.customer_id;
```