

Working with Data

Session 2 — What happens when tables meet?

Block 3 — Joins: what is one row now?

Knowledge check 2 — 10 minutes, closed book, 3 questions

- Covers Session 1 and Homework 1: the key test, `COUNT`, `NULL` in a `WHERE`, `GROUP BY`, `ORDER BY` and `LIMIT`
- Laptops closed, phones away. On the desk: the paper, a pen, and your one handwritten A4 sheet.
- One scenario, three questions. Each question stands on its own.
- Worth 5% of the final grade
- **This block's lecture is 35 minutes**, not 45: the check comes out of the lecture, never out of the lab
- Hand it in when the timer goes. Then I say the three answers, once.

Knowledge check 2 — the three answers, said once

Listen, and mark your own memory of what you wrote.

The written key is on Moodle next Monday, with the marking by clause. Your marked paper is at the TA office from the same day.

This block in one sentence

A join puts columns from two tables in one row.

After every join, two questions:

- 1. What is one row now?**
- 2. Did every row find its partner?**

What we cover

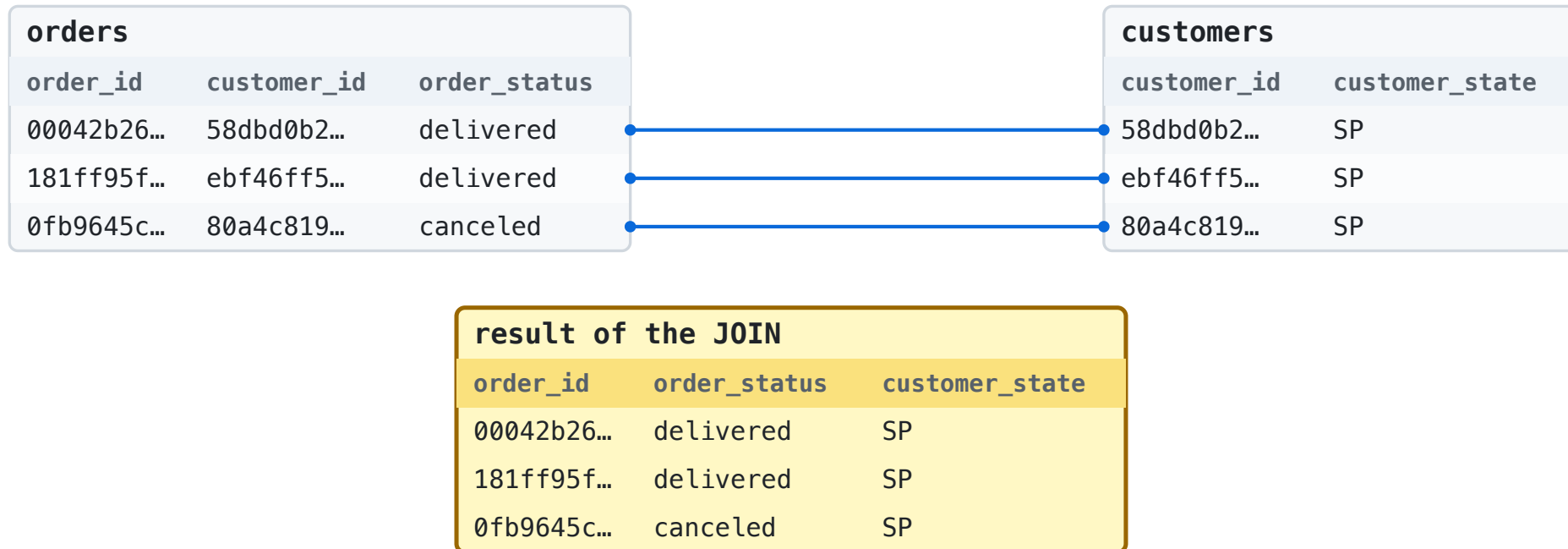
1. Why the data is in several tables, and how they point at each other
2. The first join, and the check that goes with every join
3. Join, then `GROUP BY`. A third table.
4. `JOIN` and `LEFT JOIN`: the rows that have no partner
5. Lab 3, in teams of three

Why the data is in several tables

Table	One row is	Columns we use today
orders	one order	order_id, customer_id, order_status
customers	the customer of one order	customer_id, customer_state
order_items	one item of an order	order_id, order_item_id, price
products (the lab's)	one product	product_id, product_category_name, in Portuguese
product_category_name_translation (the lab's)	one category name	its Portuguese and English spellings

An order's row does not hold its customer's state: it holds a `customer_id`, and `customers` has a row with the same `customer_id`. **The shared column is how one table points at another.**

The first join, in a picture



rows in orders: 3 · rows in the result: 3 · distinct order_id in the result: 3

Three real orders. Each one points at one customer. **Three rows in, three rows out.**

In the query, the tables get short names: `orders AS o`, `customers AS c`. Then every column says which table it comes from: `o.order_id`, `c.customer_state`.

The first join, as a query

Watch (cell 1 of `lecture.ipynb`): every order, with its customer's state.

cell 1

```
con.sql("""
    SELECT o.order_id, o.order_status, c.customer_state
    FROM orders AS o
    JOIN customers AS c ON c.customer_id = o.customer_id
    ORDER BY o.order_id
    LIMIT 5
""").df()
```

	order_id	order_status	customer_state
0	00042b26cf59d7ce69dfabb4e55b4fd9	delivered	SP
1	000e63d38ae8c00bbcb5a30573b99628	delivered	SP
2	003d804eef0e1b856881cd18e0cc0d4c	delivered	MA
3	0046e1d57f4c07c8c92ab26be8c3dfc0	delivered	SP
4	0049edf42837aa808e03552a62bd2158	delivered	RJ

FROM one table, **JOIN** the other, **ON** the columns that must be equal.

Predict, then watch: how many rows did the join return?

Cell 2:

```
con.sql("""
    SELECT COUNT(*)                AS rows_in_result,
           COUNT(DISTINCT o.order_id) AS distinct_orders_in_result
    FROM orders AS o
    JOIN customers AS c ON c.customer_id = o.customer_id
""").df()
```

Predict: `orders` has 2,653 rows (`SELECT COUNT(*) FROM orders` , Session 1), and `customers` has 2,653 rows. What are the two numbers?

How many rows did the join return? — what it printed

cell 2

```
con.sql("""
    SELECT COUNT(*)                               AS rows_in_result,
           COUNT(DISTINCT o.order_id) AS distinct_orders_in_result
    FROM orders AS o
    JOIN customers AS c ON c.customer_id = o.customer_id
""").df()
```

	rows_in_result	distinct_orders_in_result
0	2653	2653

2,653 in · 2,653 rows out · 2,653 distinct orders out. With `JOIN`, three equal counts mean every order found exactly one customer. Nothing was left out, and nothing was repeated. One row is still one order.

The three counts go with every join: the rows that went in, the rows that came out, the distinct keys that came out. This pair of tables is the easy case, one customer per order, so the three agree. They are here for the joins where they do not, in a few slides.

Join, then **GROUP BY**: orders per customer state

Watch (cell 3). Every clause from Session 1 works in a query that has a join in it: here **GROUP BY** and **ORDER BY**.

cell 3

```
con.sql("""  
    SELECT c.customer_state, COUNT(*) AS orders  
    FROM orders AS o  
    JOIN customers AS c ON c.customer_id = o.customer_id  
    GROUP BY c.customer_state  
    ORDER BY orders DESC  
    LIMIT 5  
""").df()
```

	customer_state	orders
0	SP	1133
1	RJ	348
2	MG	302
3	RS	139
4	PR	122

A third table is one more **JOIN** line

Watch (cell 4): item sales by customer state. The price is on `order_items`; the state is on `customers`; `orders` connects them: `order_id` from items to orders, `customer_id` from orders to customers. `price` is one item's amount: `order_items` has no quantity column, so `SUM(i.price)` is the total. Last week it was `Quantity * Price`.

```
cell 4 con.sql("""
    SELECT c.customer_state,
           COUNT(*)           AS items,
           ROUND(SUM(i.price), 2) AS item_sales
    FROM order_items AS i
    JOIN orders AS o    ON o.order_id = i.order_id
    JOIN customers AS c ON c.customer_id = o.customer_id
    GROUP BY c.customer_state
    ORDER BY item_sales DESC
    LIMIT 5
    """).df()
```

	customer_state	items	item_sales
0	SP	1278	143900.82
1	RJ	432	53806.44
2	MG	338	35744.61
3	RS	149	24200.28
4	PR	139	14404.21

The same check, for three tables

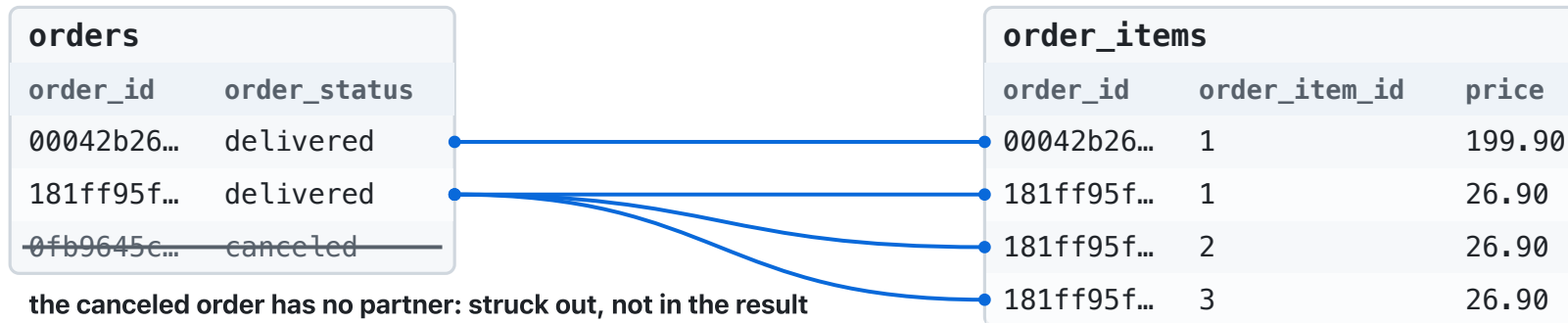
Watch (cell 5). An item is identified by two columns, so the distinct count takes the pair.

```
cell 5 con.sql("""  
    SELECT COUNT(*) AS rows_in_result,  
           COUNT(DISTINCT (i.order_id, i.order_item_id)) AS distinct_items_in_result  
    FROM order_items AS i  
    JOIN orders AS o ON o.order_id = i.order_id  
    JOIN customers AS c ON c.customer_id = o.customer_id  
""").df()
```

	rows_in_result	distinct_items_in_result
0	3024	3024

3,024 items in · 3,024 rows out · 3,024 distinct items out. Every item kept, exactly once. One row is one item.

A join where some rows have no partner



result of the JOIN			
order_id	order_status	order_item_id	price
00042b26...	delivered	1	199.90
181ff95f...	delivered	1	26.90
181ff95f...	delivered	2	26.90
181ff95f...	delivered	3	26.90

rows in orders: 3 · rows in the result: 4 · distinct order_id in the result: 2

The same three orders, joined to their items. **One has one item. One has three. One has none.**

Predict, then watch: every order with its items

Cell 6:

```
con.sql("""
    SELECT COUNT(*)                AS rows_in_result,
           COUNT(DISTINCT o.order_id) AS distinct_orders_in_result
    FROM orders AS o
    JOIN order_items AS i ON i.order_id = o.order_id
""").df()
```

Predict: `orders` has 2,653 rows. Will the result have more rows than that, or fewer? And will all 2,653 orders be in it?

Every order with its items — what it printed

cell 6

```
con.sql("""
    SELECT COUNT(*)                               AS rows_in_result,
           COUNT(DISTINCT o.order_id) AS distinct_orders_in_result
    FROM orders AS o
    JOIN order_items AS i ON i.order_id = o.order_id
""").df()
```

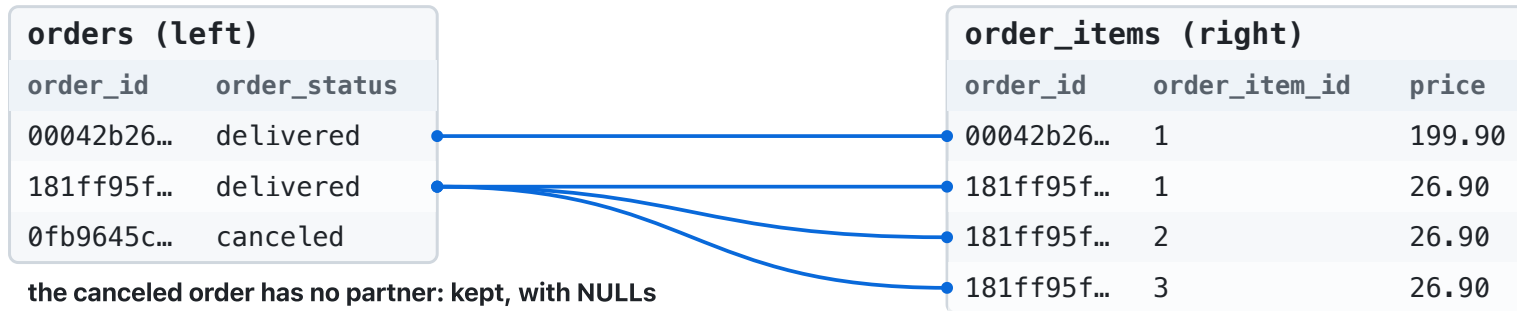
	rows_in_result	distinct_orders_in_result
0	3024	2610

2,653 in · 3,024 rows out · 2,610 distinct orders out.

More rows: 3,024. One row is now one item, and `order_items` has 3,024 rows.

Fewer orders: 2,610. 43 orders have no items. `JOIN` keeps only the rows that found a partner, and says nothing about the rest.

LEFT JOIN keeps every row of the left table



result of the LEFT JOIN			
order_id	order_status	order_item_id	price
00042b26...	delivered	1	199.90
181ff95f...	delivered	1	26.90
181ff95f...	delivered	2	26.90
181ff95f...	delivered	3	26.90
0fb9645c...	canceled	NULL	NULL

rows in orders: 3 · rows in the result: 5 · distinct order_id in the result: 3

FROM orders AS o LEFT JOIN order_items AS i : every order stays. Where an order has no partner, the item columns are NULL .

Predict, then watch: the same query, with `LEFT JOIN` instead of `JOIN`

Cell 7: one word added.

```
con.sql("""
    SELECT COUNT(*)                AS rows_in_result,
           COUNT(DISTINCT o.order_id) AS distinct_orders_in_result
    FROM orders AS o
    LEFT JOIN order_items AS i ON i.order_id = o.order_id
""").df()
```

Predict: with `JOIN` : 2,653 orders in, 3,024 rows out, 2,610 distinct orders out. What are the two numbers now?

The same query, with **LEFT JOIN** — what it printed

cell 7

```
con.sql("""
    SELECT COUNT(*)                               AS rows_in_result,
           COUNT(DISTINCT o.order_id) AS distinct_orders_in_result
    FROM orders AS o
    LEFT JOIN order_items AS i ON i.order_id = o.order_id
""").df()
```

	rows_in_result	distinct_orders_in_result
0	3067	2653

2,653 in · 3,067 rows out · 2,653 distinct orders out. 3,067 is 3,024 and 43 more: one row for each order that has no items. Every row of the left table, **orders**, is there at least once; where it has no partner, the columns from **order_items** are **NULL**.

What **LEFT JOIN** kept

Watch (cell 8): the three orders of the picture, in the result of the **LEFT JOIN**.

```
cell 8 con.sql("""
    SELECT left(o.order_id, 8) AS order_id, o.order_status, i.order_item_id, i.price
    FROM orders AS o
    LEFT JOIN order_items AS i ON i.order_id = o.order_id
    WHERE left(o.order_id, 8) IN ('00042b26', '181ff95f', '0fb9645c')
    ORDER BY o.order_id, i.order_item_id
""").df()
```

	order_id	order_status	order_item_id	price
0	00042b26	delivered	1	199.9
1	0fb9645c	canceled	<NA>	NaN
2	181ff95f	delivered	1	26.9
3	181ff95f	delivered	2	26.9
4	181ff95f	delivered	3	26.9

<NA>, NaN : pandas's **NULL**. `left(..., 8)` : a text function, not **LEFT JOIN**.

The rows with no partner

Watch (cell 9): `LEFT JOIN`, then keep the rows where the right table's key `IS NULL`: the column from the other table is `NULL` exactly when there was no partner.

cell 9

```
con.sql("""  
    SELECT COUNT(*) AS orders_with_no_items  
    FROM orders AS o  
    LEFT JOIN order_items AS i ON i.order_id = o.order_id  
    WHERE i.order_id IS NULL  
""").df()
```

	orders_with_no_items
0	43

The rows with no partner: what are they?

Watch (cell 10): the same join and the same rows. What changes is the question: which orders are they? Grouped by the order's status.

cell 10

```
con.sql("""
    SELECT o.order_status, COUNT(*) AS orders
    FROM orders AS o
    LEFT JOIN order_items AS i ON i.order_id = o.order_id
    WHERE i.order_id IS NULL
    GROUP BY o.order_status
    ORDER BY orders DESC
""").df()
```

	order_status	orders
0	canceled	25
1	unavailable	17
2	invoiced	1

A **NULL** key finds no partner

Watch (cell 11): cell 8's five rows, with one more condition: `AND i.order_item_id = i.order_item_id`. A value is always equal to itself, so every real row passes. A `NULL` is not a value: `NULL = NULL` is not true, so that row fails.

```
cell 11 con.sql("""
    SELECT left(o.order_id, 8) AS order_id, o.order_status, i.order_item_id, i.price
    FROM orders AS o
    LEFT JOIN order_items AS i ON i.order_id = o.order_id
    WHERE left(o.order_id, 8) IN ('00042b26', '181ff95f', '0fb9645c')
        AND i.order_item_id = i.order_item_id
    ORDER BY o.order_id, i.order_item_id
    """).df()
```

	order_id	order_status	order_item_id	price
0	00042b26	delivered	1	199.9
1	181ff95f	delivered	1	26.9
2	181ff95f	delivered	2	26.9
3	181ff95f	delivered	3	26.9

Which join does the question need?

The question is about...	Write
only the rows that have a partner	<code>JOIN</code>
every row of one table, with or without a partner	<code>FROM</code> that table, then <code>LEFT JOIN</code>
the rows that have no partner	<code>LEFT JOIN</code> , then <code>WHERE</code> the right table's key <code>IS NULL</code>

Put the table whose rows you must keep after `FROM` , then `LEFT JOIN` the other one.

The names you will meet elsewhere: `JOIN` is an **inner join** (`INNER JOIN`); `LEFT JOIN` is an **outer join** (`LEFT OUTER JOIN`). `RIGHT JOIN` keeps every row of the right table, `FULL JOIN` every row of both: you will read them, and rarely need them.

What you just saw, as one idea

A join **runs, and prints a table**, whether or not every row found a partner.

- `orders` joined to `customers` : 2,653 in · 2,653 rows out · 2,653 distinct orders out. Every row found one partner.
- `orders` joined to `order_items` : 2,653 in · 3,024 rows out · 2,610 distinct orders out. One row became one item, and 43 orders are not there.

Nothing in the result says which of these happened. The three counts do, and so does a total computed without the join.

From today, labs are done in teams of three

One team for the whole day. **One work laptop.** Three roles:

Role	What you do
Keyboard	Type on the work laptop, and only what the team has agreed
AI	Use an assistant on your own device. Nobody sees that screen, and you never type on the work laptop. For the first five minutes you share nothing from it.
Checker	Before a query runs, say what must stay the same, go up or go down. Afterwards, compare. Read each check out.

In the second lab you take a role you have not had today. The homework, the knowledge check and the exam are yours alone.

5-minute stand-and-stretch — start the Lab 3 clone first

The keyboard person starts the clone and `uv sync`. Then everyone takes the break: it downloads while you are away.

In your terminal (Git Bash on Windows, Terminal on macOS), in the folder where you keep course work:

```
git clone https://github.com/earino/ecbs5294-lab03-joins.git  
cd ecbs5294-lab03-joins  
uv sync
```

Then leave the laptop. Back in five.

Lab 3 — The category table that came out short

Where this lab goes: you write your own join, and find out what this marketplace sells most. Then, in section D, a colleague's table of item sales by product category that adds up to **351,299.08 BRL**, where total item sales are **364,110.54 BRL**. Nothing errors.

First minute: the clone finished? The keyboard opens the folder in VS Code, opens `notebooks/report.ipynb`, picks the `.venv` kernel, types your names in the *Your team* cell, and runs the three setup cells. Then top to bottom, **running each cell as you come to it**.

Your work, in `report.ipynb`. **A:** look at the two tables, and name the column they share. **B:** your first join, from an empty cell, with its check. **C:** what sells most? Guess, then write it. **D:** your colleague's report: count what its join kept, find the items that are missing, then keep them, and finish the one comment: why the table came out short. **E:** the check, supplied: five lines, `PASS` or `FAIL`.

The syntax is on one page: `REFERENCE-sql.md`. The team next to you at minute 33. The checkpoint is the notebook, on Moodle.

The last ten minutes — explain it to the team next to you

Two teams, one laptop at a time. At your laptop, finished or not, start with one result you found interesting; then the last step you reached; if you repaired the report, why it was wrong and what the check printed. About two minutes.

The other team asks one question. If you reached the report: **why was it wrong, not just where?** If not: **what is one row of what you showed?** Then swap laptops.

Then the what-if, both teams together: **a teammate writes the repair with the two tables the other way round: `FROM product_category_name_translation AS t LEFT JOIN order_items_w_cat` . Does that table add up to the items total? Which rows does it keep?** Commit to an answer before anyone runs it. Unsure, or the teams disagree? **Hands up.**

The lab ends here, finished or not. Before you leave, the keyboard uploads the notebook as it is to Moodle; everyone else pastes the *Your team* line.