

Working with Data

Session 2 — What happens when tables meet?

Block 4 — When one row has several partners

This block in one sentence

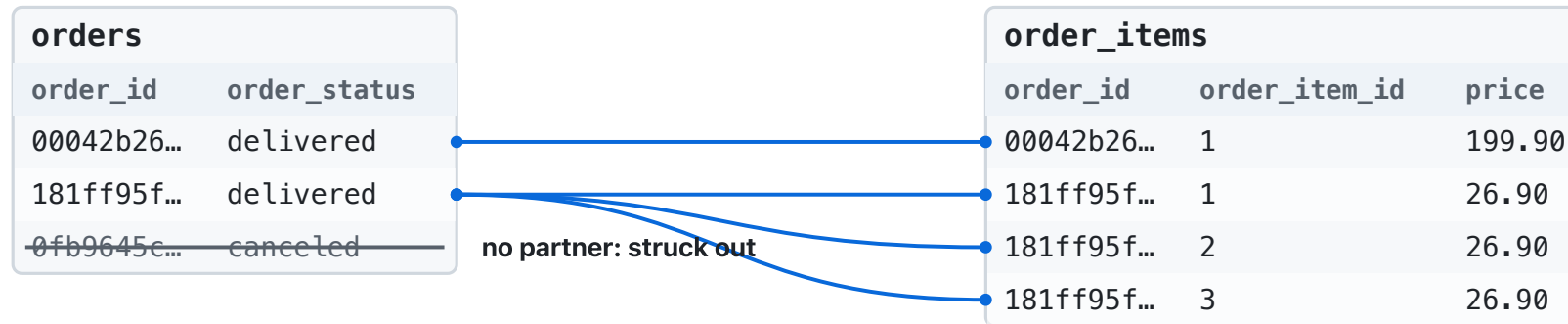
After `orders JOIN order_items`, one row is one item, not one order. So `COUNT(*)` counts items, and `AVG` averages over items. Or make the rows match your question first.

The query runs either way. Only one sentence tells a right number from a wrong one: *what is one row now?*

Agenda

1. Block 3's picture, once more
2. One join, two questions: right for one, wrong for the other
3. What is one row now?
4. **HAVING** : finding the keys that repeat
5. The repair: one row per order first, with **WITH** , then join
6. Why **DISTINCT** is a patch
7. Homework 2 · Lab 4, in your teams, with new roles

Block 3's picture: one order, three rows



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

Three real orders from São Paulo. The order with three items is **three rows** of the result. The canceled order has no row at all.

Orders per customer state

Watch (cell 1): Block 3's cell 3. `orders` joined to `customers`, then `GROUP BY`.

```
cell 1 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

Before the `GROUP BY`, one row is still **one order**: every order has exactly one customer.

The same count, after one more join

Predict, then watch (cell 2): cell 1, with one more **JOIN** line.

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

Predict: cell 1 printed 1,133 for **SP**, São Paulo. Is that number the same now, larger, or smaller?

The same count, after one more join — what it printed

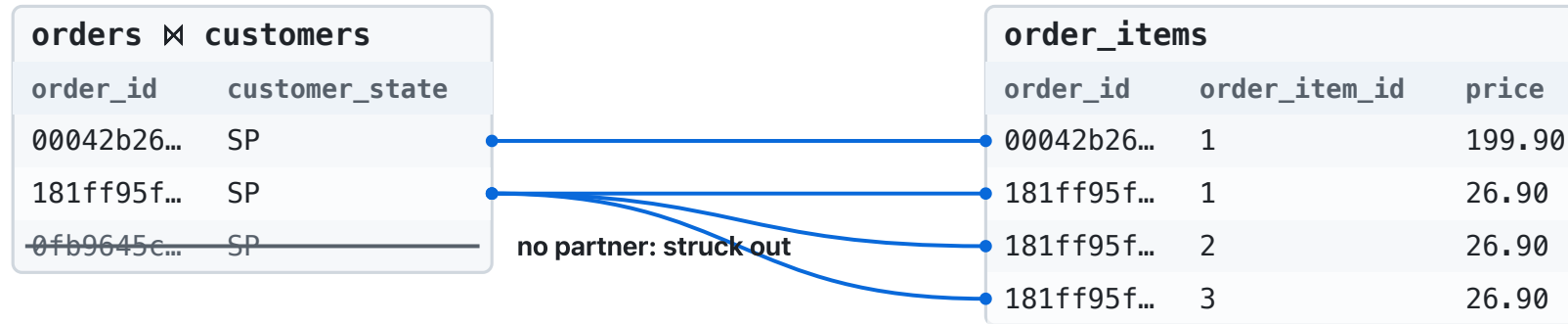
cell 2

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

	customer_state	orders
0	SP	1278
1	RJ	432
2	MG	338
3	RS	149
4	PR	139

Before the **GROUP BY**, one row is one item now. The table has one row per customer state, and **COUNT(*)** counted the rows that went into each: São Paulo's 1,278 items, in a column called **orders**.

What is one row now?



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

orders in SP: 3 · rows for SP after the join: 4 · COUNT(*) now counts items

A join sets what one row is; what you count decides the answer. One row is one item here, so COUNT(*) gives 4. *Items in São Paulo: 4. Orders: 3.* And the join dropped the canceled order, so COUNT(DISTINCT order_id) gives 2: a different wrong answer.

One order's rows in that join

Watch (cell 3): the order with three items, `181ff95f`, in cell 2's join, before its `GROUP BY`.

```
cell 3 con.sql("""
    SELECT left(o.order_id, 8) AS order_id, c.customer_state, i.order_item_id, i.price
    FROM orders AS o
    JOIN customers AS c ON c.customer_id = o.customer_id
    JOIN order_items AS i ON i.order_id = o.order_id
    WHERE left(o.order_id, 8) = '181ff95f'
    ORDER BY i.order_item_id
""").df()
```

	order_id	customer_state	order_item_id	price
0	181ff95f	SP	1	26.9
1	181ff95f	SP	2	26.9
2	181ff95f	SP	3	26.9

One order, three rows. `COUNT(*)` counts it three times; `AVG` would average over its three items.

HAVING: a condition on a group

Watch (cell 4): which orders have more than one item?

```
cell 4 con.sql("""
    SELECT order_id, COUNT(*) AS items
    FROM order_items
    GROUP BY order_id
    HAVING COUNT(*) > 1
    ORDER BY items DESC, order_id
    LIMIT 5
    """).df()
```

	order_id	items
0	ab14fdcfbe524636d65ee38360e22ce8	20
1	be382a9e1ed25128148b97d6bfdb21af	8
2	62073ec6b54b8e6322037fc0f3591ad3	7
3	28cf49bbfe2683759d3cf866b7fda22f	6
4	3e7bc2221b2c46fafa06f741386784fc	6

Session 1's rule: after **GROUP BY**, every column is grouped or aggregated, so a count exists only for a group. A condition on a count needs the groups first: that is **HAVING**.

FROM → **WHERE** → **GROUP BY** → **HAVING** → **SELECT** → **ORDER BY** → **LIMIT**

The same condition in **WHERE**

Predict, then watch (cell 5): cell 4's condition, moved to where Session 1 put every condition.

```
con.sql("""
    SELECT order_id, COUNT(*) AS items
    FROM order_items
    WHERE COUNT(*) > 1
    GROUP BY order_id
""").df()
```

Predict: does it run?

The same condition in **WHERE** — what it printed

cell 5

```
try:
    con.sql("""
        SELECT order_id, COUNT(*) AS items
        FROM order_items
        WHERE COUNT(*) > 1
        GROUP BY order_id
        """).df()
except duckdb.Error as e:
    print("DuckDB refused:", str(e).splitlines()[0])
```

DuckDB refused: Binder Error: WHERE clause cannot contain aggregates!

WHERE filters **rows**, before any group exists. One row has no count: a count belongs to a group. **HAVING** filters **groups**, after.

How many orders have more than one item

Watch (cell 6): cell 4's query without its `LIMIT 5`, with a count around it.

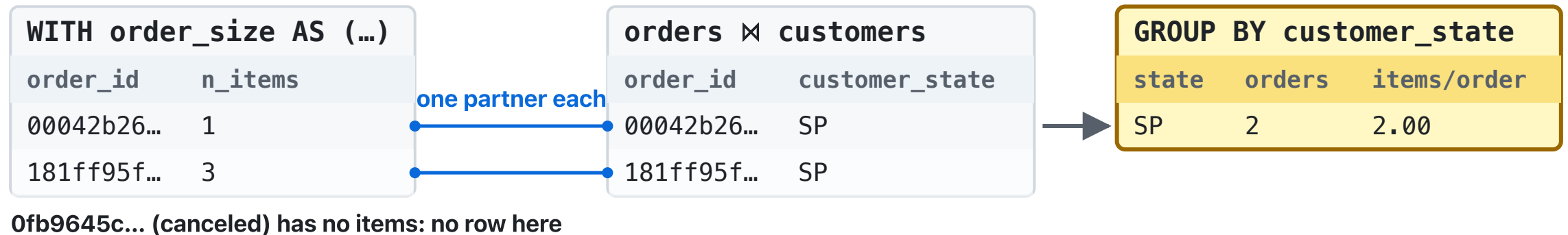
```
cell 6 row = con.sql("""
    SELECT COUNT(*) AS orders_that_repeat, SUM(items) AS their_item_rows
    FROM (SELECT order_id, COUNT(*) AS items
          FROM order_items
          GROUP BY order_id
          HAVING COUNT(*) > 1)
    """).fetchone()                                # the first row of the result; [0] and [1] are its two columns
print("orders that repeat:", row[0], " item rows they hold:", row[1])
```

```
orders that repeat: 265    item rows they hold: 679
```

265 orders hold 679 item rows: 414 more rows than orders. An order with three items is ordinary; what the join does with it is not.

The repair: one row per order first

One row per order first, then join: every row finds exactly one partner



rows going in: 2 · rows coming out: 2 · distinct orders: 2

Make the table with several rows per order **one row per order**, in a `WITH` named `order_size`: each order's id and its number of items. Then every row finds exactly one partner, and `COUNT(*)` counts orders again. The canceled order has no items, so it has no row in `order_size`: three slides on, that matters.

The **WITH** query on its own

Watch (cell 7): `WITH name AS ( ... )` names a query. The query below it reads `name` like a table. Its name in every tutorial: a **common table expression**, a CTE.

cell 7

```
con.sql("""
    WITH order_size AS (
        SELECT order_id, COUNT(*) AS n_items
        FROM order_items
        GROUP BY order_id
    )
    SELECT COUNT(*)                AS rows_in_order_size,
           COUNT(DISTINCT order_id) AS distinct_orders
    FROM order_size
""").df()
```

	rows_in_order_size	distinct_orders
0	2610	2610

2,610 rows, 2,610 different orders: in `order_size`, one row is one order.

Then join

Predict, then watch (cell 8): `order_size` ,joined through `orders` to `customers` .

```
con.sql("""
    WITH order_size AS (
        SELECT order_id, COUNT(*) AS n_items
        FROM order_items
        GROUP BY order_id
    )
    SELECT c.customer_state, COUNT(*) AS orders,
           ROUND(AVG(s.n_items), 2) AS items_per_order
    FROM order_size AS s
    JOIN orders AS o    ON o.order_id = s.order_id
    JOIN customers AS c ON c.customer_id = o.customer_id
    GROUP BY c.customer_state
    ORDER BY orders DESC LIMIT 5
""").df()
```

New here is only `FROM order_size AS s` : the joins, the `GROUP BY` and the `ORDER BY` are cell 1's.

Predict: `SP` 's `orders` this time: 1,133, larger, or smaller?

Then join — what it printed

cell 8

```
con.sql("""
    WITH order_size AS (
        SELECT order_id, COUNT(*) AS n_items
        FROM order_items
        GROUP BY order_id
    )
    SELECT c.customer_state, COUNT(*) AS orders,
           ROUND(AVG(s.n_items), 2) AS items_per_order
    FROM order_size AS s
    JOIN orders AS o    ON o.order_id = s.order_id
    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	items_per_order
0	SP	1116	1.15
1	RJ	342	1.26
2	MG	295	1.15
3	RS	137	1.09
4	PR	121	1.15

COUNT(*) counts orders again. 1,116, not 1,133: São Paulo's 17 orders with no items have no row in `order_size`.

The check for that join

Watch (cell 9): the three counts, as after every join.

```
cell 9 con.sql("""
    WITH order_size AS (
        SELECT order_id, COUNT(*) AS n_items
        FROM order_items
        GROUP BY order_id
    )
    SELECT (SELECT COUNT(*) FROM order_size) AS rows_going_in,
           COUNT(*)                        AS rows_coming_out,
           COUNT(DISTINCT s.order_id)      AS distinct_orders_out
    FROM order_size AS s
    JOIN orders AS o   ON o.order_id = s.order_id
    JOIN customers AS c ON c.customer_id = o.customer_id
""").df()
```

	rows_going_in	rows_coming_out	distinct_orders_out
0	2610	2610	2610

All three equal, with **JOIN**: every row found exactly one partner. Nothing was repeated, and nothing was lost, by this join.

The same **WITH**, adding up money

Watch (cell 10): `order_items_total` is one row per order with what its items cost; then its average by state. The lab does the same with payments.

```
cell 10 con.sql("""
    WITH order_items_total AS (
        SELECT order_id, SUM(price) AS item_sales
        FROM order_items
        GROUP BY order_id
    )
    SELECT c.customer_state,
           COUNT(*)           AS orders,
           ROUND(AVG(v.item_sales), 2) AS avg_item_sales
    FROM order_items_total AS v
    JOIN orders AS o      ON o.order_id = v.order_id
    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	avg_item_sales
0	SP	1116	128.94
1	RJ	342	157.33
2	MG	295	121.17
3	RS	137	176.64
4	PR	121	119.04

Inside the **WITH**, the aggregate the question needs: **COUNT** for how many, **SUM** for how much. São Paulo: 128.94 per order.

DISTINCT patches one column, not the table

Watch (cell 11): São Paulo after the join to its items, as in cell 2.

```
cell 11 con.sql("""
    SELECT COUNT(*)                AS rows_in_join,
           COUNT(DISTINCT o.order_id) AS orders,
           ROUND(SUM(i.price), 2)    AS item_sales,
           ROUND(AVG(i.price), 2)    AS avg_price
    FROM orders AS o
    JOIN customers AS c  ON c.customer_id = o.customer_id
    JOIN order_items AS i ON i.order_id = o.order_id
    WHERE c.customer_state = 'SP'
""").df()
```

	rows_in_join	orders	item_sales	avg_price
0	1278	1116	143900.82	112.6

SUM 143,900.82: right. **COUNT(*)** 1,278: wrong for orders; **DISTINCT** counts the orders with items, 1,116. **AVG** 112.60: per item, and **no change to the count fixes it**. Per order is 128.94 (cell 10).

The rule

1. **Say what one row is:** of each table, and of the result of the join.
2. **Start from the table whose rows are what the question counts.**
3. If the number lives on a table with several rows per order, **add it up to one row per order first**, in a `WITH` . Then join.
4. **Three counts after every join.**

Homework 2 assigned — Tables that meet: three numbers for the board

The board wants **three numbers** from the Olist files, answered **alone**. About **3 hours**, as a budget. Due **Saturday 17 October, 23:59**, on Moodle.

The starter is `earino/ecbs5294-hw02-joins` on GitHub; its README has the clone line, for home. It opens with a worked example.

1. **Revenue by month** (*a sum*)
2. **How many orders have no review, by order status** (*a sum: with and without add up to all orders*)
3. **For each customer state: the paid orders, the revenue, the average order value** (*a ratio*)

Under every question: the query, its result, the check cell, and one sentence. The check cell's `print` lines are written; you write the question's query, and fill the check's blanks: the three counts of your join, and the same number without it. **The notebook is the whole submission**, uploaded to Moodle.

5-minute stand-and-stretch — start the Lab 4 clone first

Roles change for Lab 4. **The new keyboard person clones**, on the laptop that will be the team's work laptop:

```
git clone https://github.com/earino/ecbs5294-lab04-repeated-rows.git  
cd ecbs5294-lab04-repeated-rows  
uv sync
```

Then stand up. Five minutes away from the screen.

Lab 4 — More orders than there are orders

Same team, new roles. Keyboard · AI (own device; the assistant's output waits five minutes; never types here) · Checker (says what must stay equal, rise or fall before each query runs; reads each check out).

First minute: the clone finished? Open the folder in VS Code, open `notebooks/report.ipynb`, pick the `.venv` kernel, type your names in the *Your team* cell, and run the three setup cells. Then top to bottom, **running each cell as you come to it.**

Your work, in `report.ipynb`. **A:** five rows of `order_payments`, the orders by status, and one count. **B:** your join, from an empty cell: what was paid, by order status, with its check; say what one row is. **C:** you write it: the orders with more than one payment row, as a view, and a join that shows their payments. **D:** a colleague's report asks section B's join a different question; then the repair, the one query you compose, with one comment: why the report counted 2,790. **E:** the supplied check: five numbered lines, `PASS` or `FAIL`; then say what one row of your table is, and read its rows.

In section D: the report runs without an error and counts **2,790 paid orders**. `orders.csv` has **2,653 rows**. A paid order is an order with at least one payment row, so paid orders cannot be more than orders. 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: **orders . csv** has one more delivered order than the repaired table. Is that table wrong? Which query would show the missing one? 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.