Module 6 of 16

Aggregation

COUNT, SUM, AVG, MIN and MAX and how they treat NULL; GROUP BY on columns and expressions; HAVING versus WHERE; conditional aggregation for counts, pivots and rates; GROUP_CONCAT; and the grouping mistakes — ONLY_FULL_GROUP_BY, the three COUNTs and averages of averages.

Most questions asked of a database are not about single rows but about many at once: how many orders came in, the revenue per category, the months that beat a target, the approval rate per country, the products sold each day. Aggregation is how SQL answers them. Aggregate functions collapse rows into a value, `GROUP BY` does it once per group, and `HAVING` keeps the groups that matter. It is the heart of reporting and analytics SQL, and the topic behind a large share of interview questions — duplicates, monthly reports, pivots, "count by status".

Aggregate functions covers `COUNT`, `SUM`, `AVG`, `MIN` and `MAX`, the three forms of `COUNT`, NULLs being skipped, and what an empty input returns. GROUP BY covers grouping by columns, by several columns and by expressions such as the month, the rule for the SELECT list, NULL groups and the logical order of a grouped query. HAVING covers filtering groups, how it differs from `WHERE`, and the duplicate finder. Conditional aggregation covers `SUM(CASE …)`, pivoting rows into columns and rates with `AVG`. GROUP_CONCAT covers `DISTINCT`, `ORDER BY` and `SEPARATOR` inside the call, the length limit and `STRING_AGG` elsewhere. Grouping pitfalls covers error 1055 and `ONLY_FULL_GROUP_BY`, `ANY_VALUE`, the counts, `WHERE` versus `HAVING`, averages of averages and NULL totals.

The exercises are single-table reports: an order summary, ratings with gaps, revenue per category, sign-ups per month and plan, loyal customers, duplicate emails, orders by status per month, approval rates, products sold by date, team rosters, the three counts per department and a weighted unit price. The checkpoint adds a monthly sales summary with a floor, the classic monthly transactions report and a daily product list.

CONTENTS

1	Aggregate functions — COUNT, SUM, AVG, MIN and MAX	13 min
2	GROUP BY — one summary row per group	14 min
3	HAVING — filtering groups	12 min

4	Conditional aggregation — SUM(CASE …) and pivots	14 min
5	GROUP_CONCAT — a group's values as one string	12 min
6	Grouping pitfalls — ONLY_FULL_GROUP_BY, COUNT and averages	14 min
✓	Checkpoint — Aggregation	25 min

Exercises in this module compile on MySQL 8 syntax on SQLite 3.49. Reference solutions and the answer key follow each lesson.

LESSON 1 · 13 MIN

Aggregate functions — COUNT, SUM, AVG, MIN and MAX

Every query so far has returned one output row per input row, possibly filtered and sorted. Many questions are not like that. *How many orders came in? What was the revenue? What is the average rating, the cheapest product, the most recent sign-up?* Each answer is a single value computed from many rows. That is what aggregate functions do, and they are the foundation of every report, dashboard and analytics question — and of a large share of SQL interview questions. This lesson covers the five standard aggregates on a whole table; the next adds `GROUP BY` to compute them per group.

The five aggregates

```sql
SELECT COUNT(*)     AS orders,
       SUM(amount)  AS revenue,
       AVG(amount)  AS average_order,
       MIN(amount)  AS smallest,
       MAX(amount)  AS largest
FROM orders;
```

orders	revenue	average_order	smallest	largest
5	750	150	40	400

Five rows went in, one row came out. That is the defining property: an aggregate collapses its input rows into one value, so a query whose SELECT list holds only aggregates (and no `GROUP BY`) returns exactly **one row** — even when the table is empty.

- `COUNT` counts.
- `SUM` adds numbers.
- `AVG` is `SUM` divided by the number of values it summed.
- `MIN` and `MAX` find the smallest and largest. They also work on text (alphabetically first and last) and dates (earliest and latest): `MAX(order_date)` is the most recent order.

Aggregates and WHERE

`WHERE` runs **before** aggregation: it decides which rows the aggregates see.

```sql
SELECT COUNT(*) AS march_orders, SUM(amount) AS march_revenue
FROM orders
WHERE order_date ⩾ '2024-03-01' AND order_date < '2024-04-01';
```

The reverse is not allowed: an aggregate cannot appear in `WHERE`, because at that stage the rows have not been aggregated yet. `WHERE SUM(amount) > 1000` is an error; the clause for conditions on aggregates is `HAVING`, two lessons on.

The three COUNTs

`COUNT` comes in three forms, and interviewers love the difference:

```sql
SELECT COUNT(*)                    AS all_rows,
       COUNT(coupon_code)          AS with_coupon,
       COUNT(DISTINCT customer_id) AS customers
FROM orders;
```

- `COUNT(*)` counts **rows**. NULLs do not matter; a row is a row.
- `COUNT(expr)` counts rows where `expr` is **not NULL**.
- `COUNT(DISTINCT expr)` counts the different non-NULL values.

`COUNT(1)` is identical to `COUNT(*)` — the 1 is never NULL — and there is no performance difference in modern MySQL. A favourite trick question: "what does `COUNT(NULL)` return?" Always 0, because NULL is never counted.

NULLs are skipped

Every aggregate except `COUNT(*)` **ignores NULL**. That is usually what you want, and occasionally a trap.

Take five reviews where two have no rating:

id	rating
1	4
2	NULL
3	5
4	NULL
5	3

```sql
SELECT COUNT(*)           AS reviews,          -- 5
       COUNT(rating)      AS rated,            -- 3
       SUM(rating)        AS total,            -- 12
       AVG(rating)        AS avg_rated,        -- 4      (12 / 3)
       SUM(rating) / COUNT(*)       AS avg_all,           -- 2.4    (12 / 5)
       AVG(COALESCE(rating, 0))     AS avg_missing_as_0   -- 2.4
FROM reviews;
```

`AVG(rating)` is the average *of the reviews that have a rating*. Whether that or 2.4 is right depends on what a missing rating means — "not given" (skip it) or "zero stars" (count it). The query has to say which; `COALESCE` inside the aggregate is how you say the second.

Empty input

When no rows reach the aggregates — an empty table or a `WHERE` that matches nothing — the query still returns one row:

```sql
SELECT COUNT(*) AS n, SUM(amount) AS total, AVG(amount) AS avg_amount, MAX(amo
FROM orders
WHERE amount > 1000000;
```

n	total	avg_amount	biggest
0	NULL	NULL	NULL

`COUNT` is 0, but `SUM`, `AVG`, `MIN` and `MAX` are **NULL**, not 0 — there is no total of nothing. A revenue report that prints NULL for a quiet day is usually a bug in the report;

`COALESCE(SUM(amount), 0)` fixes it. The same NULL appears when every value in the column is NULL.

AVG and integer columns

MySQL's `AVG` of an integer column returns a decimal: the average of 1, 2 and 2 is `1.6667`. PostgreSQL also returns a decimal for `AVG`. SQL Server, however, returns an **integer** average for an integer column — `AVG` of 1, 2, 2 is 1 there — which is why SQL Server queries often read `AVG(score * 1.0)` or `AVG(CAST(score AS DECIMAL(10, 2)))`. Round averages for display: `ROUND(AVG(score), 2)`.

`AVG(DISTINCT col)` and `SUM(DISTINCT col)` also exist and use each different value once. They are rarely what a question means — two orders of 100 are two orders, not one.

Aggregating expressions

The argument of an aggregate can be any expression, evaluated per row before aggregating:

```
SELECT SUM(price * qty)                AS revenue,
       MAX(DATEDIFF(shipped_on, ordered_on)) AS slowest_days,
       COUNT(DISTINCT DATE(created_at))      AS active_days,
       MIN(CHAR_LENGTH(name))                AS shortest_name
FROM order_lines;
```

And an aggregate can sit inside an expression, evaluated after aggregating: `ROUND(100 * SUM(refunded) / COUNT(*), 1)` is a refund percentage, and `MAX(price) - MIN(price)` is the price range.

Mixing aggregates with plain columns

A query that aggregates the whole table returns one row, so what should a plain column next to an aggregate show?

```
SELECT name, MAX(amount) FROM orders;      -- which order's name?
```

There are five names and one row. Standard SQL rejects this query, and so does MySQL in its default mode (`ONLY_FULL_GROUP_BY`, the subject of this module's last lesson). Older MySQL versions accepted it and returned the name of an *arbitrary* row — not necessarily the one with the largest amount. Finding "the order with the largest amount" is a different question, answered with `ORDER BY amount DESC LIMIT 1` or, in later modules, a subquery or window function.

Pitfalls

- `COUNT(col)` skipping NULLs when you meant to count rows.
- `AVG` silently ignoring NULLs that should have been zeros.
- `SUM` of no rows is NULL, not 0.
- An aggregate in `WHERE` — use `HAVING`.
- A plain column next to aggregates without `GROUP BY`.
- SQL Server's integer `AVG` on integer columns.

Key takeaways

- `COUNT`, `SUM`, `AVG`, `MIN` and `MAX` collapse many rows into one value.
- Only `COUNT(*)` counts NULLs; every other aggregate skips them.
- `COUNT(DISTINCT x)` counts different values.
- With no input rows, `COUNT` is 0 and the others are NULL.
- `WHERE` filters rows before aggregation; it cannot test an aggregate.

EXERCISE 1

An order summary

Summarise the whole `orders` table in one row.

id	customer_id	amount	coupon
1	10	120.00	SAVE10
2	11	80.00	NULL
3	10	45.00	NULL
4	12	300.00	WELCOME
5	11	55.00	SAVE10

Return `orders` (the number of orders), `customers` (how many different customers ordered), `with_coupon` (orders that used a coupon), `revenue` (the total amount), `avg_order` (the average amount, rounded to 2 decimals) and `largest` (the biggest amount). The query always returns exactly one row — also for an empty table.

```
orders | customers | with_coupon | revenue | avg_order | largest
5 | 3 | 3 | 600 | 120 | 300
```

INPUT 1

```
CREATE TABLE orders (id INT PRIMARY KEY,
customer_id INT, amount DECIMAL(10,2),
coupon VARCHAR(20));
INSERT INTO orders VALUES (1, 10,
120.00, 'SAVE10'), (2, 11, 80.00, NULL),
(3, 10, 45.00, NULL), (4, 12, 300.00,
'WELCOME'), (5, 11, 55.00, 'SAVE10');
```

OUTPUT 1

```
orders | customers | with_coupon |
revenue | avg_order | largest
5 | 3 | 3 | 600 | 120 | 300
```

REFERENCE SOLUTION

```sql
SELECT COUNT(*)                    AS orders,
       COUNT(DISTINCT customer_id) AS customers,
       COUNT(coupon)               AS with_coupon,
       SUM(amount)                 AS revenue,
       ROUND(AVG(amount), 2)       AS avg_order,
       MAX(amount)                 AS largest
FROM orders;
```

EXERCISE 2

Ratings with gaps

Some reviews have no rating. Summarise the reviews of the product `lamp` only.

id	product	rating
1	lamp	4
2	lamp	NULL
3	desk	2
4	lamp	5
5	lamp	NULL
6	lamp	3
7	desk	5

Return one row: `reviews` (all lamp reviews), `rated` (those with a rating), `avg_rated` (the average of the given ratings), `avg_all` (the average counting a missing rating as 0) — both averages rounded to 2 decimals — and `lowest` (the lowest rating given).

```
reviews | rated | avg_rated | avg_all | lowest
5 | 3 | 4 | 2.4 | 3
```

INPUT 1

```
CREATE TABLE reviews (id INT PRIMARY
KEY, product VARCHAR(20), rating INT);
INSERT INTO reviews VALUES (1, 'lamp',
4), (2, 'lamp', NULL), (3, 'desk', 2),
(4, 'lamp', 5), (5, 'lamp', NULL), (6,
'lamp', 3), (7, 'desk', 5);
```

OUTPUT 1

```
reviews | rated | avg_rated | avg_all |
lowest
5 | 3 | 4 | 2.4 | 3
```

REFERENCE SOLUTION

```sql
SELECT COUNT(*)                          AS reviews,
       COUNT(rating)                     AS rated,
       ROUND(AVG(rating), 2)             AS avg_rated,
       ROUND(AVG(COALESCE(rating, 0)), 2) AS avg_all,
       MIN(rating)                       AS lowest
FROM reviews
WHERE product = 'lamp';
```

QUIZ

1. A column `bonus` holds 100, NULL, 200, NULL. What are `COUNT(*)`, `COUNT(bonus)` and `AVG(bonus)`?

A. 4, 4 and 75

B. 4, 2 and 150

C. 2, 2 and 150

D. 4, 2 and 75

2. `SELECT SUM(amount), COUNT(*) FROM orders WHERE amount > 1000000;` matches no rows. What does it return?

A. No rows

B. One row: `0 | 0`

C. One row: `NULL | 0`

D. One row: `NULL | NULL`

3. What does `COUNT(DISTINCT customer_id)` count on an orders table?

A. Orders

B. Orders with a customer

C. Different customers who ordered

D. Customers with exactly one order

4. Why does `SELECT customer_id FROM orders WHERE SUM(amount) > 500 GROUP BY customer_id;` fail?

A. `SUM` only works on integer columns

B. `WHERE` runs before aggregation, so it cannot use an aggregate; the condition belongs in `HAVING`

C. `GROUP BY` must come before `WHERE`

D. `customer_id` must be aggregated too

ANSWER KEY · AGGREGATE FUNCTIONS — COUNT, SUM, AVG, MIN AND MAX

1. B `COUNT(*)` counts all four rows; `COUNT(bonus)` and `AVG(bonus)` skip the NULLs, so the average is 300 / 2 = 150. Getting 75 needs `AVG(COALESCE(bonus, 0))`.

2. C An aggregate query without `GROUP BY` always returns one row. `COUNT` of nothing is 0, but `SUM` of nothing is NULL — wrap it in `COALESCE(SUM(amount), 0)` for a 0.

3. C `DISTINCT` inside `COUNT` counts each different non-NULL value once — the number of different customers. `COUNT(customer_id)` would be orders that have a customer.

4. B `WHERE` filters individual rows before any group exists. Conditions on aggregates go in `HAVING`, after `GROUP BY`. The clause order shown is the correct one.

LESSON 2 · 14 MIN

GROUP BY — one summary row per group

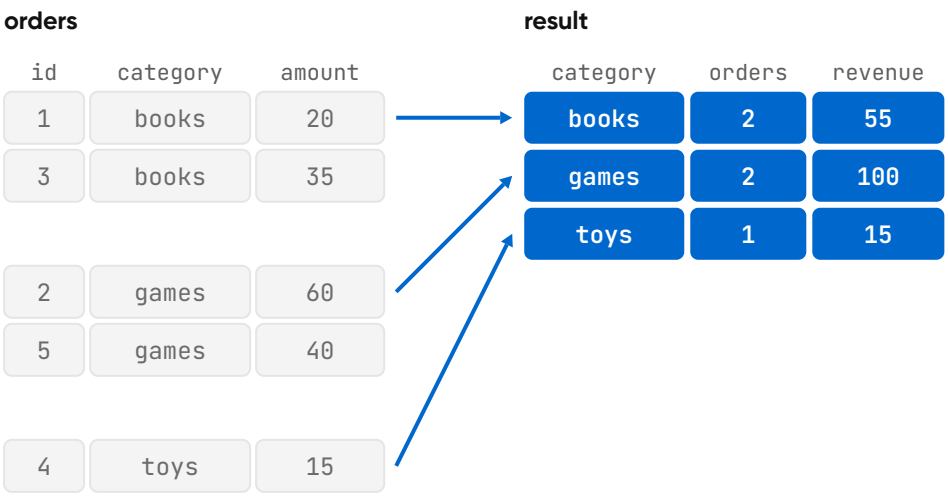
The previous lesson aggregated a whole table into one row. Real questions are almost always *per something*: revenue **per category**, orders **per customer**, sign-ups **per month**, average salary **per department**. `GROUP BY` is how SQL says "per". It splits the rows into groups that share a value, runs the aggregates once inside each group, and returns one row per group. Together with the aggregates it is the most used feature in analytical SQL.

One group per value

```sql
SELECT category,
       COUNT(*)     AS orders,
       SUM(amount)  AS revenue
FROM orders
GROUP BY category
ORDER BY category;
```

Picture the rows sorted into piles, one pile per category; then `COUNT(*)` and `SUM(amount)` run on each pile separately:

GROUP BY: one pile per value, one row per pile



1. The 5 orders as FROM and WHERE hand them on: each row on its own, categories mixed.
2. GROUP BY category puts the rows into piles, one per distinct category: books (2), games (2), toys (1). A category no order has makes no pile.
3. The aggregates run once inside each pile: COUNT(*) counts its rows and SUM(amount) adds its amounts.
4. Each pile becomes one result row: 5 rows in, 3 out. Only the grouping column and the aggregates have one value per pile, so only they can be selected.

Five rows in, three rows out — one per distinct category. A grouped query returns as many rows as there are groups, and a group exists only if at least one row has that value. A category with no orders does not appear at all; listing every category, including the empty ones, needs a join against a categories table (Module 7).

The rule for the SELECT list

Once rows are grouped, each output row stands for a whole group. So everything in the `SELECT` list must have **one value per group**:

- a column listed in `GROUP BY` — it is the same for every row in the group;
- an aggregate — it computes one value from the group;
- an expression built from those two.

```
SELECT category, COUNT(*), MAX(amount), ROUND(AVG(amount), 2)    -- fine
FROM orders GROUP BY category;

SELECT category, customer_id, COUNT(*)                           -- which cus
FROM orders GROUP BY category;
```

The second query asks for one `customer_id` per category when a category has many. Standard SQL rejects it; MySQL 8 rejects it too by default, with error 1055 — the `ONLY_FULL_GROUP_BY` rule, covered in detail in the grouping-pitfalls lesson.

Grouping by several columns

List several columns and a group is each distinct **combination**:

```sql
SELECT region, category, SUM(amount) AS revenue
FROM orders
GROUP BY region, category
ORDER BY region, category;
```

region	category	revenue
north	books	20
north	games	100
south	books	35
south	toys	15

Only combinations that occur are returned — there is no `north, toys` row because no such order exists. The order of columns inside `GROUP BY` does not change the groups; `ORDER BY` decides how they are printed.

Grouping by an expression

You can group by any expression, which is how dates are rolled up into periods:

```sql
SELECT DATE_FORMAT(order_date, '%Y-%m') AS month,
       COUNT(*)                         AS orders,
       SUM(amount)                      AS revenue
FROM orders
GROUP BY DATE_FORMAT(order_date, '%Y-%m')
ORDER BY month;
```

MySQL also lets you name the alias in `GROUP BY` — `GROUP BY month` — which saves repeating the expression. PostgreSQL allows that too; SQL Server does not, so portable code repeats the expression. Other useful groupings:

```
GROUP BY YEAR(order_date), MONTH(order_date)   -- year and month as numbers
GROUP BY DAYNAME(order_date)                    -- sales by weekday, every wee
GROUP BY FLOOR(age / 10) * 10                   -- age bands: 20, 30, 40…
GROUP BY amount ⩾ 100                           -- two groups: 0 (small) and 1
GROUP BY LOWER(TRIM(email))                     -- addresses that differ only
```

Grouping by `MONTH(order_date)` alone puts March 2023 and March 2024 in the same group — sometimes intended (seasonality), usually a bug.

WHERE before GROUP BY

`WHERE` filters rows **before** they are grouped, so it changes what each group contains:

```
SELECT category, COUNT(*) AS big_orders
FROM orders
WHERE amount ⩾ 50
GROUP BY category
ORDER BY category;
```

Categories with no order of at least 50 disappear entirely, because none of their rows reaches the grouping. That is often what you want; when it is not — "every category, with its count of big orders, zero included" — use conditional aggregation (two lessons on) instead of `WHERE`.

The logical order of a grouped query is worth memorising, since many interview questions test it:

1. `FROM` — the rows
2. `WHERE` — keep some rows
3. `GROUP BY` — form groups
4. `HAVING` — keep some groups (next lesson)
5. `SELECT` — compute the output columns
6. `ORDER BY` — sort
7. `LIMIT` — cut

Sorting groups

MySQL 5.7 and earlier sorted the result of `GROUP BY` by the grouping columns as a side effect; MySQL 8 does not, and no other database promises it. Always write `ORDER BY`. You can sort by an aggregate, which is how "top" questions are answered:

```sql
SELECT customer_id, SUM(amount) AS spent
FROM orders
GROUP BY customer_id
ORDER BY spent DESC, customer_id
LIMIT 3;
```

The second sort key breaks ties, so the top three is the same list every time.

NULL forms one group

`GROUP BY` puts **all NULLs together in one group**, even though `NULL = NULL` is not true in a `WHERE`. Orders with no region come out as one row with region NULL:

```sql
SELECT COALESCE(region, 'unknown') AS region, COUNT(*) AS orders
FROM orders
GROUP BY COALESCE(region, 'unknown')
ORDER BY region;
```

Labelling the NULL group makes a report readable; leaving it out silently is a common way for totals not to add up.

GROUP BY versus DISTINCT

With no aggregates, `GROUP BY` returns the distinct combinations, exactly like `SELECT DISTINCT`:

```sql
SELECT DISTINCT region FROM orders;
SELECT region FROM orders GROUP BY region;      -- same rows
```

Use `DISTINCT` to say "unique values" and `GROUP BY` when you compute something per group. Readers expect it that way round.

Pitfalls

- A plain column in SELECT that is not in `GROUP BY`.
- Expecting groups with no rows to appear.
- Grouping by `MONTH(d)` when the report spans years.
- Relying on `GROUP BY` to sort.
- Filtering with `WHERE` when every group should still appear.
- Forgetting that NULLs form their own group.

Key takeaways

- `GROUP BY` returns one row per distinct value (or combination) of its columns.
- Select only grouping columns, aggregates, and expressions of them.
- Group by expressions like `DATE_FORMAT(d, '%Y-%m')` to roll dates into periods.
- `WHERE` runs before grouping; `ORDER BY` after — always write it.
- All NULLs fall into one group.

EXERCISE 1

Revenue per category

Report on `orders` per category. Orders with no category belong to a group labelled `uncategorised`.

id	category	amount
1	books	20
2	games	60
3	books	35
4	toys	15
5	games	40
6	toys	85

Return `category`, `orders`, `revenue` and `avg_amount` (rounded to 2 decimals). Order by `revenue` descending, then `category`.

```
category | orders | revenue | avg_amount
games | 2 | 100 | 50
toys | 2 | 100 | 50
books | 2 | 55 | 27.5
```

INPUT 1

```
CREATE TABLE orders (id INT PRIMARY KEY,
category VARCHAR(20), amount INT);
INSERT INTO orders VALUES (1, 'books',
20), (2, 'games', 60), (3, 'books', 35),
(4, 'toys', 15), (5, 'games', 40), (6,
'toys', 85);
```

OUTPUT 1

```
category | orders | revenue | avg_amount
games | 2 | 100 | 50
toys | 2 | 100 | 50
books | 2 | 55 | 27.5
```

REFERENCE SOLUTION

```sql
SELECT COALESCE(category, 'uncategorised') AS category,
       COUNT(*)                            AS orders,
       SUM(amount)                         AS revenue,
       ROUND(AVG(amount), 2)               AS avg_amount
FROM orders
GROUP BY COALESCE(category, 'uncategorised')
ORDER BY revenue DESC, category;
```

EXERCISE 2

Sign-ups per month and plan

Count sign-ups per calendar month and plan.

id	signed_up_on	plan
1	2024-01-05	free
2	2024-01-20	pro
3	2024-01-31	free
4	2024-02-01	free
5	2024-02-14	pro
6	2024-02-29	pro
7	2023-02-10	free

Return `month` (`2024-01`), `plan` and `signups`. February 2023 and February 2024 are different months. Order by `month`, then `plan`.

```
month | plan | signups
2023-02 | free | 1
2024-01 | free | 2
2024-01 | pro | 1
2024-02 | free | 1
2024-02 | pro | 2
```

INPUT 1

```
CREATE TABLE signups (id INT PRIMARY
KEY, signed_up_on DATE, plan
VARCHAR(10));
INSERT INTO signups VALUES (1, '2024-01-
05', 'free'), (2, '2024-01-20', 'pro'),
(3, '2024-01-31', 'free'), (4, '2024-02-
01', 'free'),
  (5, '2024-02-14', 'pro'), (6, '2024-
02-29', 'pro'), (7, '2023-02-10',
'free');
```

OUTPUT 1

```
month | plan | signups
2023-02 | free | 1
2024-01 | free | 2
2024-01 | pro | 1
2024-02 | free | 1
2024-02 | pro | 2
```

REFERENCE SOLUTION

```
SELECT DATE_FORMAT(signed_up_on, '%Y-%m') AS month,
       plan,
       COUNT(*)                           AS signups
FROM signups
GROUP BY DATE_FORMAT(signed_up_on, '%Y-%m'), plan
ORDER BY month, plan;
```

QUIZ

1. `orders` has 8 rows in 3 categories. How many rows does `SELECT category, COUNT(*) FROM orders GROUP BY category` return?

A. 8

B. 3

C. 1

D. It depends on the counts

2. Which SELECT list is valid with `GROUP BY department` under standard SQL rules?

A. `department, name, COUNT(*)`

B. `department, COUNT(*), MAX(salary)`

C. `name, MAX(salary)`

D. `*`

3. A table's `region` column is NULL on four rows. How do they appear after `GROUP BY region`?

A. They are left out

B. As four separate groups

C. As one group with region NULL

D. As part of the first group

4. Does MySQL 8 return grouped rows sorted by the grouping columns?

A. Yes, always

B. No — the order is not guaranteed; write ORDER BY

C. Only for numeric columns

D. Only when the column has an index

ANSWER KEY · GROUP BY — ONE SUMMARY ROW PER GROUP

1. B A grouped query returns one row per group — one per distinct category, so 3. Without `GROUP BY` the aggregate would return 1 row.

2. B Only grouping columns, aggregates and expressions of them have one value per group. `name` has many values inside a department, so any list containing it bare is ambiguous; `*` includes every column.

3. C `GROUP BY` treats all NULLs as one group, even though `NULL = NULL` is not true in a `WHERE`. Label it with `COALESCE(region, 'unknown')` in both SELECT and GROUP BY.

4. B MySQL 5.7 and earlier sorted implicitly; MySQL 8 removed that, and the SQL standard never promised it. A result whose order matters needs an explicit `ORDER BY`, ideally with a tie-breaker.

LESSON 3 · 12 MIN

HAVING — filtering groups

`GROUP BY` produced one row per group. Very often only some groups matter: customers with at least three orders, products that sold more than 100 units, departments whose average salary is above a threshold, email addresses that appear more than once. The condition is about the *group* — a count or a total — not about any single row, so `WHERE` cannot express it. `HAVING` can. It is the `WHERE` of groups.

Filtering on an aggregate

```
SELECT customer_id, COUNT(*) AS orders, SUM(amount) AS spent
FROM orders
GROUP BY customer_id
HAVING COUNT(*) ⩾ 3
ORDER BY spent DESC, customer_id;
```

The rows are grouped by customer, the aggregates are computed per customer, and then `HAVING` throws away every group whose count is below 3. What survives is printed.

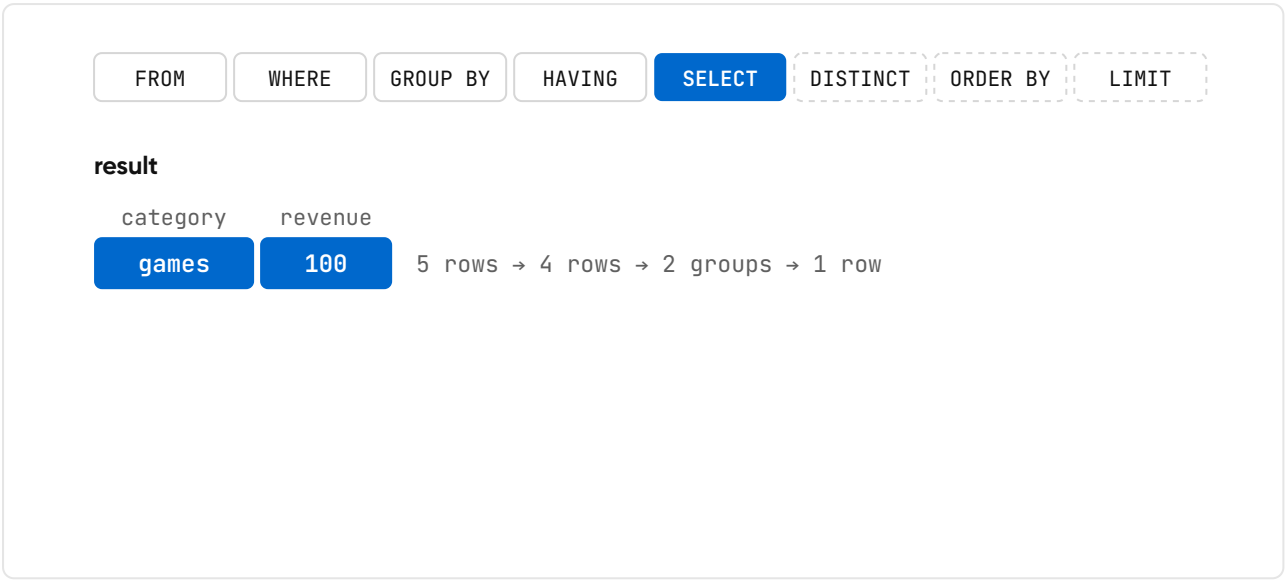
Trying the same with `WHERE` fails:

```
SELECT customer_id, COUNT(*) FROM orders
WHERE COUNT(*) ⩾ 3          -- error: invalid use of group function
GROUP BY customer_id;
```

At `WHERE` time each row is still on its own, and "the count of this row's group" does not exist yet.

Where HAVING sits in the order of execution

WHERE filters rows before grouping; HAVING filters groups after



1. FROM hands on all 5 orders. Two filters follow, and the pipeline across the top says when each one runs.
2. WHERE runs before any grouping and sees one row at a time, so it can only test that row's own columns. The toys order of 15 fails and never reaches a group.
3. GROUP BY piles the 4 surviving rows by category. Only now do totals exist — which is why WHERE SUM(amount) > 60 would be an error.
4. HAVING runs after grouping and sees each finished group with its aggregates. books adds up to 55, not more than 60, so the whole group goes.
5. SELECT prints one row per surviving group. Row conditions belong in WHERE, where they shrink the work early; HAVING is for conditions only a group can answer.

- `WHERE` sees individual rows and decides which ones are grouped.
- `HAVING` sees finished groups, with their aggregates, and decides which are kept.

Both can appear in one query, each doing its own job:

```
SELECT product_id, SUM(qty) AS units
FROM order_lines
WHERE order_date ⩾ '2024-01-01' AND order_date < '2025-01-01'    -- rows: 202
GROUP BY product_id
HAVING SUM(qty) > 100                                            -- groups: 
ORDER BY units DESC, product_id;
```

Read it as: *of the 2024 order lines, group by product, keep the products that sold more than 100 units*.

The classic: finding duplicates

"Find the duplicate emails" is one of the most asked SQL questions, and it is a `GROUP BY` with a `HAVING`:

```sql
SELECT email, COUNT(*) AS copies
FROM users
GROUP BY email
HAVING COUNT(*) > 1
ORDER BY email;
```

Group by the value that should be unique; any group with more than one row is a duplicate. For duplicates across several columns — the same person entered twice with the same name and birth date — group by all of them: `GROUP BY first_name, last_name, birth_date HAVING COUNT(*) > 1`. When the duplicates differ in case or spacing, normalise in the grouping: `GROUP BY LOWER(TRIM(email))`.

The same shape answers "values that appear exactly once" (`HAVING COUNT(*) = 1`), "customers who ordered from at least three categories" (`HAVING COUNT(DISTINCT category) ⩾ 3`) and "days with no failed jobs" (`HAVING SUM(status = 'failed') = 0`).

Conditions in HAVING

`HAVING` takes any condition that has one value per group — aggregates, grouping columns, and combinations of them with `AND`, `OR` and `NOT`:

```sql
HAVING COUNT(*) ⩾ 3 AND SUM(amount) > 500
HAVING MAX(amount) > 2 * AVG(amount)              -- a group with an outlier
HAVING MIN(order_date) ⩾ '2024-01-01'             -- customers whose first ord
HAVING COUNT(DISTINCT product_id) = 1             -- customers who only ever l
```

The aggregate in `HAVING` does not have to be in the SELECT list. `HAVING AVG(rating) ⩾ 4` filters on the average even if only the names are printed.

Put row conditions in WHERE

A condition on a grouping column works in either clause:

```sql
SELECT region, SUM(amount) FROM orders WHERE region <> 'test' GROUP BY region;
SELECT region, SUM(amount) FROM orders GROUP BY region HAVING region <> 'test
```

The results are the same, but the first is better: it removes the rows before grouping, so less work is done, and it tells the reader the condition is about rows. The rule of thumb: **if a condition does not need an aggregate, put it in `WHERE`**. Interviewers ask "what is the difference between WHERE and HAVING?" expecting exactly this: `WHERE` filters rows before grouping and cannot use aggregates; `HAVING` filters groups after grouping and can.

Some conditions are not interchangeable at all. "Customers whose orders are all above 50" is a group property:

```
HAVING MIN(amount) > 50
```

`WHERE amount > 50` would instead drop the small orders and keep every customer who has *any* order above 50 — a different question with a different answer.

Aliases in HAVING

MySQL lets `HAVING` refer to a SELECT alias:

```
SELECT customer_id, SUM(amount) AS spent
FROM orders
GROUP BY customer_id
HAVING spent > 500;            -- MySQL only
```

Standard SQL, PostgreSQL and SQL Server reject that, because `HAVING` is evaluated before `SELECT` names its columns. Writing the aggregate again — `HAVING SUM(amount) > 500` — works everywhere and costs nothing: the database computes it once.

HAVING without GROUP BY

With no `GROUP BY`, the whole table is a single group, and `HAVING` decides whether its one row is returned:

```
SELECT COUNT(*) AS late_orders
FROM orders
WHERE shipped_on > due_on
HAVING COUNT(*) > 0;
```

This returns a row only if there are late orders. It is legal and occasionally useful in alerting queries, but rare.

Pitfalls

- An aggregate in `WHERE` — it belongs in `HAVING`.
- Row conditions in `HAVING` that should be in `WHERE`.
- `WHERE amount > 50` when the question means "every order of the group is above 50" (`HAVING MIN(amount) > 50`).
- Relying on SELECT aliases in `HAVING` in a database that is not MySQL.
- Forgetting `ORDER BY` after filtering groups.

Key takeaways

- `HAVING` filters groups; `WHERE` filters rows.
- Only `HAVING` can test aggregates such as `COUNT(*)` and `SUM(x)`.
- Duplicates are `GROUP BY key HAVING COUNT(*) > 1`.
- Put conditions that need no aggregate in `WHERE`.
- MySQL accepts aliases in `HAVING`; portable SQL repeats the aggregate.

EXERCISE 1

Loyal customers

Find customers with **at least 3 paid orders** whose paid orders total **more than 200**. Orders with any other status do not count at all.

id	customer_id	amount	status
1	1	50	paid
2	1	80	paid
3	1	90	paid
4	2	300	paid
5	2	10	refunded
6	2	20	paid
7	3	100	paid
8	3	60	paid
9	3	70	paid
10	3	5	paid
11	4	10	paid
12	4	20	paid
13	4	30	paid

Return `customer_id`, `orders` and `spent`. Order by `spent` descending, then `customer_id`.

```
customer_id | orders | spent
3 | 4 | 235
1 | 3 | 220
```

INPUT 1

```
CREATE TABLE orders (id INT PRIMARY KEY,
customer_id INT, amount INT, status
VARCHAR(10));
INSERT INTO orders VALUES (1, 1, 50,
'paid'), (2, 1, 80, 'paid'), (3, 1, 90,
'paid'), (4, 2, 300, 'paid'), (5, 2, 10,
'refunded'),
  (6, 2, 20, 'paid'), (7, 3, 100,
'paid'), (8, 3, 60, 'paid'), (9, 3, 70,
'paid'), (10, 3, 5, 'paid'),
  (11, 4, 10, 'paid'), (12, 4, 20,
'paid'), (13, 4, 30, 'paid');
```

OUTPUT 1

```
customer_id | orders | spent
3 | 4 | 235
1 | 3 | 220
```

REFERENCE SOLUTION

```sql
SELECT customer_id,
       COUNT(*)    AS orders,
       SUM(amount) AS spent
FROM orders
WHERE status = 'paid'
GROUP BY customer_id
HAVING COUNT(*) ⩾ 3 AND SUM(amount) > 200
ORDER BY spent DESC, customer_id;
```

EXERCISE 2

Duplicate emails

Find the email addresses that appear more than once in `users`. Addresses that differ only in case or surrounding spaces are the same address, so compare `LOWER(TRIM(email))`.

id	email
1	ada@x.com
2	bob@y.com
3	`'Ada@X.com '`
4	cy@z.com
5	bob@y.com
6	bob@y.com

Return `email` (normalised), `copies` and `first_id` (the smallest id holding it). Order by `email`.

```
email | copies | first_id
ada@x.com | 2 | 1
bob@y.com | 3 | 2
```

INPUT 1

```
CREATE TABLE users (id INT PRIMARY KEY,
email VARCHAR(80));
INSERT INTO users VALUES (1,
'ada@x.com'), (2, 'bob@y.com'), (3,
'Ada@X.com '), (4, 'cy@z.com'), (5,
'bob@y.com'), (6, 'bob@y.com');
```

OUTPUT 1

```
email | copies | first_id
ada@x.com | 2 | 1
bob@y.com | 3 | 2
```

REFERENCE SOLUTION

```sql
SELECT LOWER(TRIM(email)) AS email,
       COUNT(*)           AS copies,
       MIN(id)            AS first_id
FROM users
GROUP BY LOWER(TRIM(email))
HAVING COUNT(*) > 1
ORDER BY email;
```

QUIZ

1. Which query finds the customers whose **every** order is above 100?

A. `… WHERE amount > 100 GROUP BY customer_id`

B. `… GROUP BY customer_id HAVING MIN(amount) > 100`

C. `… GROUP BY customer_id HAVING SUM(amount) > 100`

D. `… GROUP BY customer_id HAVING MAX(amount) > 100`

2. Which clause should hold `region <> 'test'` in a query that groups by region and filters on `SUM(amount) > 1000`?

A. `HAVING`, next to the SUM condition

B. `WHERE`

C. Either, and it makes no difference at all

D. `ORDER BY`

3. What does this return when `orders` has 50 rows?

```sql
SELECT COUNT(*) AS n FROM orders HAVING COUNT(*) > 100;
```

A. One row: `50`

B. One row: `0`

C. No rows

D. An error: HAVING needs GROUP BY

4. `SELECT customer_id, SUM(amount) AS total FROM orders GROUP BY customer_id HAVING total > 500;` runs in MySQL. Where else does it work unchanged?

   A. PostgreSQL

   B. SQL Server

   C. Both

   D. Neither — portable code writes `HAVING SUM(amount) > 500`

ANSWER KEY · HAVING — FILTERING GROUPS

1. B If the smallest order of a customer is above 100, all of them are. The `WHERE` version keeps anyone with *some* order above 100 (and so does `MAX`); `SUM` tests the total.

2. B The result is the same in either clause, but a condition that needs no aggregate belongs in `WHERE`: it removes the rows before grouping (less work) and says the condition is about rows. The `SUM` condition can only go in `HAVING`.

3. C Without `GROUP BY` the whole table is one group; `HAVING` discards it because 50 is not above 100, so the result is empty. `HAVING` without `GROUP BY` is legal.

4. D Using a SELECT alias in `HAVING` is a MySQL extension; standard SQL evaluates `HAVING` before the SELECT list names its columns. Repeating the aggregate works everywhere.

LESSON 4 · 14 MIN

Conditional aggregation — SUM(CASE …) and pivots

`WHERE` and `GROUP BY` answer one question per query: *how many paid orders per month?* Reports usually want several answers side by side — paid, pending and refunded orders per month, in three columns, with a zero where there were none. Running three queries and gluing the results together is slow and clumsy. **Conditional aggregation** does it in one pass: put the condition *inside* the aggregate. It is the technique behind pivot tables, rates and percentages, and a large family of interview questions (monthly transaction reports, approval rates, "count by status").

The idea: an aggregate over a CASE

An aggregate runs over the values of an expression. Make the expression 1 when a row qualifies and 0 when it does not, and `SUM` counts the qualifying rows:

```sql
SELECT DATE_FORMAT(created_on, '%Y-%m') AS month,
       COUNT(*) AS orders,
       SUM(CASE WHEN status = 'paid'     THEN 1 ELSE 0 END) AS paid,
       SUM(CASE WHEN status = 'pending'  THEN 1 ELSE 0 END) AS pending,
       SUM(CASE WHEN status = 'refunded' THEN 1 ELSE 0 END) AS refunded
FROM orders
GROUP BY DATE_FORMAT(created_on, '%Y-%m')
ORDER BY month;
```

month	orders	paid	pending	refunded
2024-01	4	3	0	1
2024-02	3	1	2	0

Every row of every month goes through all three `CASE`s; each row adds 1 to exactly one column. The month with no pending orders shows 0 — it did not disappear, which is what a `WHERE status = 'pending'` would have done to it.

Three spellings of a conditional count

```
SUM(CASE WHEN status = 'paid' THEN 1 ELSE 0 END)    -- standard, everywhere
COUNT(CASE WHEN status = 'paid' THEN 1 END)         -- standard: no ELSE → NULL
SUM(status = 'paid')                                -- MySQL only: a comparison
```

The `COUNT` form relies on `CASE` returning NULL when no branch matches and `COUNT(expr)` skipping NULLs. Writing `ELSE 0` there would be a bug — `COUNT` counts zeros. The MySQL shortcut is compact and common in MySQL code, but PostgreSQL rejects `SUM` of a boolean and SQL Server has no boolean expressions in that position, so use `CASE` in anything meant to travel. Note one difference: if `status` is NULL, `SUM(status = 'paid')` adds NULL (skipped), while the `CASE` form adds 0 — the totals are the same.

Conditional sums of amounts

Return the amount instead of 1 and you get conditional totals:

```
SELECT customer_id,
       SUM(CASE WHEN status = 'paid'     THEN amount ELSE 0 END) AS paid_amoun
       SUM(CASE WHEN status = 'refunded' THEN amount ELSE 0 END) AS refunded_a
       SUM(CASE WHEN status = 'paid' THEN amount ELSE 0 END)
     - SUM(CASE WHEN status = 'refunded' THEN amount ELSE 0 END) AS net
FROM orders
GROUP BY customer_id
ORDER BY customer_id;
```

Use `ELSE 0` here rather than leaving it out: without it a customer with no refunds gets `NULL` for `refunded_amount`, and `net` becomes NULL too, because any arithmetic with NULL is NULL.

Pivoting rows into columns

Turning the values of one column into separate output columns is called a **pivot**. Quarterly sales stored one row per sale become one row per product with a column per quarter:

```sql
SELECT product,
       SUM(CASE WHEN QUARTER(sold_on) = 1 THEN amount ELSE 0 END) AS q1,
       SUM(CASE WHEN QUARTER(sold_on) = 2 THEN amount ELSE 0 END) AS q2,
       SUM(CASE WHEN QUARTER(sold_on) = 3 THEN amount ELSE 0 END) AS q3,
       SUM(CASE WHEN QUARTER(sold_on) = 4 THEN amount ELSE 0 END) AS q4
FROM sales
WHERE sold_on ⩾ '2024-01-01' AND sold_on < '2025-01-01'
GROUP BY product
ORDER BY product;
```

The rows of the output are the `GROUP BY`; the columns are the `CASE`s. MySQL has no `PIVOT` keyword (SQL Server does, and PostgreSQL has `crosstab` in an extension), so this pattern *is* how MySQL pivots — and it works on every database. Its limitation is that the columns must be known when the query is written; a pivot over values that change needs the query built by application code.

`MAX(CASE WHEN … THEN value END)` is the pivot form for *non-numeric* values — for example turning key/value rows (`attribute = 'color'`, `value = 'red'`) into one row per item with a column per attribute. Each group has at most one matching row, and `MAX` just picks it out of the NULLs.

Rates and percentages

A rate is a conditional count divided by a count:

```sql
SELECT country,
       COUNT(*) AS trans_count,
       SUM(CASE WHEN state = 'approved' THEN 1 ELSE 0 END)  AS approved_count,
       ROUND(100 * SUM(CASE WHEN state = 'approved' THEN 1 ELSE 0 END) / COUNT
FROM transactions
GROUP BY country
ORDER BY country;
```

`AVG` does the division for you: the average of a column of 1s and 0s is the fraction of 1s.

```sql
ROUND(100 * AVG(CASE WHEN state = 'approved' THEN 1 ELSE 0 END), 1) AS approva
ROUND(AVG(rating ⩾ 4), 2)                                          -- MySQL:
```

Two cautions. In PostgreSQL and SQL Server integer division turns `SUM(…) / COUNT(*)` into 0 — write `100.0 *`. And `AVG(CASE WHEN … THEN 1 END)` without `ELSE 0` averages only the 1s

and always returns 1; the `ELSE 0` is what puts the non-matching rows in the denominator.

Conditional MIN, MAX and DISTINCT counts

Any aggregate takes a `CASE`:

```
SELECT customer_id,
       MIN(CASE WHEN status = 'paid' THEN created_on END)            AS first_
       MAX(CASE WHEN status = 'refunded' THEN created_on END)        AS last_r
       COUNT(DISTINCT CASE WHEN status = 'paid' THEN product_id END) AS produc
FROM orders
GROUP BY customer_id
ORDER BY customer_id;
```

Here the absence of `ELSE` is deliberate: non-matching rows give NULL, and `MIN`, `MAX` and `COUNT(DISTINCT …)` skip NULLs. A customer with no refunds gets NULL for `last_refund`, which is the honest answer.

Filtering on conditional aggregates

Conditional aggregates work in `HAVING` like any other:

```
SELECT customer_id
FROM orders
GROUP BY customer_id
HAVING SUM(CASE WHEN status = 'refunded' THEN 1 ELSE 0 END) = 0     -- never r
   AND COUNT(*) ⩾ 2
ORDER BY customer_id;
```

"Customers who bought A but never B" is the same shape: `HAVING SUM(CASE WHEN product = 'A' THEN 1 ELSE 0 END) > 0 AND SUM(CASE WHEN product = 'B' THEN 1 ELSE 0 END) = 0`.

Standard SQL also defines a `FILTER` clause — `COUNT(*) FILTER (WHERE status = 'paid')` — that PostgreSQL and SQLite support. MySQL does not, so in MySQL the `CASE` form is the way.

Pitfalls

- `COUNT(CASE WHEN … THEN 1 ELSE 0 END)` counts every row — drop the `ELSE` or use `SUM`.
- `AVG(CASE WHEN … THEN 1 END)` without `ELSE 0` is always 1.
- Conditional sums without `ELSE 0` turning totals into NULL.
- Integer division in rates outside MySQL.
- Using `WHERE` for one condition when every group should stay.

Key takeaways

- `SUM(CASE WHEN cond THEN 1 ELSE 0 END)` counts matching rows inside each group.
- Several conditional aggregates side by side pivot rows into columns.
- `AVG` over a 1/0 `CASE` is a rate; multiply by 100 for a percentage.
- `MIN` / `MAX` / `COUNT(DISTINCT)` over a `CASE` with no `ELSE` pick from the matching rows only.
- MySQL's `SUM(cond)` shortcut is not portable; `CASE` is.

EXERCISE 1

Orders by status, per month

Pivot `orders` into one row per month with a column per status.

id	created_on	status	amount
1	2024-01-03	paid	10
2	2024-01-09	paid	20
3	2024-01-15	refunded	30
4	2024-01-30	paid	40
5	2024-02-02	pending	50
6	2024-02-11	paid	60
7	2024-02-28	pending	70
8	2024-03-01	refunded	80

Return `month` (`2024-01`), `orders` (all orders that month), `paid`, `pending` and `refunded` (the count of each status, 0 when there are none). Any other status counts only in `orders`. Order by `month`.

```
month | orders | paid | pending | refunded
2024-01 | 4 | 3 | 0 | 1
2024-02 | 3 | 1 | 2 | 0
2024-03 | 1 | 0 | 0 | 1
```

INPUT 1

```
CREATE TABLE orders (id INT PRIMARY KEY,
created_on DATE, status VARCHAR(10),
amount INT);
INSERT INTO orders VALUES (1, '2024-01-
03', 'paid', 10), (2, '2024-01-09',
'paid', 20), (3, '2024-01-15',
'refunded', 30), (4, '2024-01-30',
'paid', 40),
  (5, '2024-02-02', 'pending', 50), (6,
'2024-02-11', 'paid', 60), (7, '2024-02-
28', 'pending', 70), (8, '2024-03-01',
'refunded', 80);
```

OUTPUT 1

```
month | orders | paid | pending |
refunded
2024-01 | 4 | 3 | 0 | 1
2024-02 | 3 | 1 | 2 | 0
2024-03 | 1 | 0 | 0 | 1
```

REFERENCE SOLUTION

```sql
SELECT DATE_FORMAT(created_on, '%Y-%m') AS month,
       COUNT(*) AS orders,
       SUM(CASE WHEN status = 'paid'     THEN 1 ELSE 0 END) AS paid,
       SUM(CASE WHEN status = 'pending'  THEN 1 ELSE 0 END) AS pending,
       SUM(CASE WHEN status = 'refunded' THEN 1 ELSE 0 END) AS refunded
FROM orders
GROUP BY DATE_FORMAT(created_on, '%Y-%m')
ORDER BY month;
```

EXERCISE 2

Approval rate by country

Each transaction is `approved` or `declined`.

id	country	state	amount
1	IN	approved	100
2	IN	declined	50
3	IN	approved	300
4	US	declined	20
5	US	approved	80
6	US	declined	40
7	DE	declined	10

Return `country`, `trans_count`, `approved_pct` — the percentage of transactions approved, rounded to 1 decimal — and `avg_approved`, the average amount of the approved transactions only (rounded to 2 decimals, NULL when there are none). Order by `approved_pct` descending, then `country`.

```
country | trans_count | approved_pct | avg_approved
IN | 3 | 66.7 | 200
US | 3 | 33.3 | 80
DE | 1 | 0 | NULL
```

INPUT 1

```
CREATE TABLE transactions (id INT
PRIMARY KEY, country VARCHAR(5), state
VARCHAR(10), amount INT);
INSERT INTO transactions VALUES (1,
'IN', 'approved', 100), (2, 'IN',
'declined', 50), (3, 'IN', 'approved',
300), (4, 'US', 'declined', 20),
  (5, 'US', 'approved', 80), (6, 'US',
'declined', 40), (7, 'DE', 'declined',
10);
```

OUTPUT 1

```
country | trans_count | approved_pct |
avg_approved
IN | 3 | 66.7 | 200
US | 3 | 33.3 | 80
DE | 1 | 0 | NULL
```

REFERENCE SOLUTION

```sql
SELECT country,
       COUNT(*) AS trans_count,
       ROUND(100 * AVG(CASE WHEN state = 'approved' THEN 1 ELSE 0 END), 1) AS approv
       ROUND(AVG(CASE WHEN state = 'approved' THEN amount END), 2)        AS avg_app
FROM transactions
GROUP BY country
ORDER BY approved_pct DESC, country;
```

QUIZ

1. What does `COUNT(CASE WHEN status = 'paid' THEN 1 ELSE 0 END)` return for a group of 5 rows, 2 of them paid?

A. 2

B. 5

C. 3

D. 0

2. Which expression gives the fraction of orders that were refunded, per group?

A. `AVG(CASE WHEN status = 'refunded' THEN 1 END)`

B. `AVG(CASE WHEN status = 'refunded' THEN 1 ELSE 0 END)`

C. `COUNT(status = 'refunded') / COUNT(*)`

D. `SUM(status) / COUNT(*)`

3. How does MySQL pivot quarterly sales into columns `q1` … `q4` ?

A. With the `PIVOT` operator

B. With `crosstab()`

C. With one `SUM(CASE WHEN QUARTER(d) = n THEN amount ELSE 0 END)` per column, grouped by the row label

D. It cannot; pivoting needs application code

4. Why use `SUM(CASE WHEN status = 'paid' THEN 1 ELSE 0 END)` instead of `WHERE status = 'paid'` with `COUNT(*)`?

A. It is the only way to count rows

B. `WHERE` removes groups with no paid rows, and only counts one status per query

C. `WHERE` cannot compare strings

D. `COUNT(*)` ignores NULLs

ANSWER KEY · CONDITIONAL AGGREGATION — SUM(CASE …) AND PIVOTS

1. B `COUNT` counts non-NULL values, and 0 is not NULL, so every row is counted: 5. Either drop the `ELSE` (non-matching rows become NULL) or use `SUM` with `ELSE 0`.

2. B The `ELSE 0` puts every row in the average, so the result is refunded / all. Without it `AVG` sees only the 1s and returns 1. `COUNT(status = 'refunded')` counts every row with a status, true or false.

3. C MySQL has no `PIVOT` (SQL Server does) and no `crosstab` (a PostgreSQL extension). Conditional aggregates are the MySQL way and work everywhere, as long as the columns are known in advance.

4. B The conditional form keeps every group — showing 0 where nothing matched — and can count several statuses side by side in one pass. With `WHERE`, a group with no paid rows vanishes.

LESSON 5 · 12 MIN

GROUP_CONCAT — a group's values as one string

Aggregates so far have returned numbers: a count, a total, an average. Sometimes the answer a report wants is the **list** itself: the products in each order, the skills of each candidate, the days a user was active, the tags on a post. `GROUP_CONCAT` is MySQL's aggregate for that — it joins a group's values into a single string. It is also behind a well-known interview question: *for each date, the number of different products sold and their names in alphabetical order*.

The basics

```
SELECT order_id,
       GROUP_CONCAT(product) AS products
FROM order_lines
GROUP BY order_id
ORDER BY order_id;
```

order_id	products
1	pen,ink,paper
2	stapler

Like every aggregate, `GROUP_CONCAT` returns one value per group. The values are joined with a **comma and no space** by default, and their order inside the string is not guaranteed — `pen,ink,paper` is just the order the rows were read in.

The full form

```
GROUP_CONCAT([DISTINCT] expr [ORDER BY sort_expr [ASC|DESC]] [SEPARATOR 'text
```

All three options go **inside** the parentheses, in that order:

```sql
SELECT order_id,
       GROUP_CONCAT(DISTINCT product ORDER BY product SEPARATOR ', ') AS produ
FROM order_lines
GROUP BY order_id
ORDER BY order_id;
```

order_id	products
1	ink, paper, pen
2	stapler

- `DISTINCT` drops repeated values within the group, so an order with two lines of `pen` lists it once.
- `ORDER BY` sorts the values inside the string. It is independent of the query's own `ORDER BY`, which sorts the groups. Always write it when the output is compared or shown to people — the stored order is not an order.
- `SEPARATOR` sets the text between values. `SEPARATOR ''` joins with nothing.

The `ORDER BY` inside can sort by a different column from the one listed — `GROUP_CONCAT(product ORDER BY added_at)` lists products in the order they were added — and by several keys, `ORDER BY qty DESC, product`.

The interview classic: products sold by date

Given a table of `(sell_date, product)` rows with repeats, return for each date the number of **different** products and their names, sorted, comma-separated:

```sql
SELECT sell_date,
       COUNT(DISTINCT product)                                    AS num_sold,
       GROUP_CONCAT(DISTINCT product ORDER BY product SEPARATOR ',') AS produc
FROM activities
GROUP BY sell_date
ORDER BY sell_date;
```

sell_date	num_sold	products
2020-05-30	3	Basketball,Headphone,T-shirt
2020-06-01	2	Bible,Pencil
2020-06-02	1	Mask

Three things make it right: `DISTINCT` in both functions (a product sold twice on one day counts once), `ORDER BY` inside `GROUP_CONCAT`, and `ORDER BY sell_date` for the rows.

Concatenating expressions

The argument can be any expression, so each item can be built from several columns:

```sql
SELECT customer_id,
       GROUP_CONCAT(CONCAT(product, ' x', qty) ORDER BY product SEPARATOR '; ')
FROM order_lines
GROUP BY customer_id
ORDER BY customer_id;
```

customer_id	basket
7	ink x2; pen x1

MySQL also accepts several arguments — `GROUP_CONCAT(product, ':', qty)` concatenates them per row — but the `CONCAT` form is clearer and behaves the same.

NULLs

`GROUP_CONCAT` skips NULL values, like the other aggregates. If every value in a group is NULL — or the group's item expression is NULL for every row, as happens when `CONCAT` meets a NULL column — the result is NULL rather than an empty string. A group of *no* rows never exists in `GROUP BY`, so that case only appears in a whole-table `GROUP_CONCAT` with a `WHERE` that matches nothing.

```sql
SELECT team, COALESCE(GROUP_CONCAT(nickname ORDER BY nickname), '(none)') AS r
FROM players
GROUP BY team
ORDER BY team;
```

The length limit

The result of `GROUP_CONCAT` is capped at `group_concat_max_len` characters, **1,024 by default**. Longer results are cut silently — MySQL only raises a warning that applications rarely read. A list of a thousand order ids will come back as roughly the first two hundred. The fix is a session setting before the query:

```sql
SET SESSION group_concat_max_len = 1000000;
```

That limit is also a hint about design: `GROUP_CONCAT` is for short, human-readable lists in reports. When a program needs the items, return them as rows and let the program group them.

Not a substitute for rows

A comma-separated list is convenient to read and awkward to use: you cannot join to it, index it or filter it reliably (`WHERE products LIKE '%pen%'` also matches `pencil`). Store data as rows, one value per row — the first rule of normalisation (Module 12) — and use `GROUP_CONCAT` only at the very end of a query, to present the result. The reverse operation, splitting a stored comma list back into rows, is much harder in MySQL than avoiding the list in the first place.

Other databases

Database	Function
MySQL, MariaDB, SQLite	`GROUP_CONCAT(x ORDER BY x SEPARATOR ', ')` (SQLite: `GROUP_CONCAT(x, ', ')`)
PostgreSQL	`STRING_AGG(x, ', ' ORDER BY x)`
SQL Server 2017+	`STRING_AGG(x, ', ') WITHIN GROUP (ORDER BY x)`
Oracle	`LISTAGG(x, ', ') WITHIN GROUP (ORDER BY x)`

`STRING_AGG` has no default separator — it must be given — and has no `DISTINCT` in SQL Server, where you de-duplicate in a subquery first. PostgreSQL also has `ARRAY_AGG`, which builds a real array rather than a string.

Pitfalls

- No `ORDER BY` inside the call, so the list's order changes between runs.
- Putting `SEPARATOR` before `ORDER BY` — it must come last.
- Forgetting `DISTINCT` when the same value appears on several rows.
- Silent truncation at 1,024 characters.
- Storing comma-separated lists instead of rows.

Key takeaways

- `GROUP_CONCAT` joins a group's values into one string, comma-separated by default.
- `DISTINCT`, `ORDER BY` and `SEPARATOR` go inside the call, in that order.
- NULLs are skipped; an all-NULL group gives NULL.
- Results are cut at `group_concat_max_len` (1,024) characters.
- PostgreSQL and SQL Server spell it `STRING_AGG`; Oracle `LISTAGG`.

EXERCISE 1

Products sold by date

The `activities` table records each sale of a product; the same product can be sold several times a day.

sell_date	product
2020-05-30	Headphone
2020-06-01	Pencil
2020-06-02	Mask
2020-05-30	Basketball
2020-06-01	Bible
2020-06-02	Mask
2020-05-30	T-Shirt

For each date return `sell_date`, `num_sold` (the number of **different** products sold) and `products` (those products, each once, in alphabetical order, joined by a comma with no space). Order by `sell_date`.

```
sell_date | num_sold | products
2020-05-30 | 3 | Basketball,Headphone,T-Shirt
2020-06-01 | 2 | Bible,Pencil
2020-06-02 | 1 | Mask
```

INPUT 1

```
CREATE TABLE activities (sell_date DATE,
product VARCHAR(20));
INSERT INTO activities VALUES ('2020-05-
30', 'Headphone'), ('2020-06-01',
'Pencil'), ('2020-06-02', 'Mask'),
('2020-05-30', 'Basketball'),
  ('2020-06-01', 'Bible'), ('2020-06-
02', 'Mask'), ('2020-05-30', 'T-Shirt');
```

OUTPUT 1

```
sell_date | num_sold | products
2020-05-30 | 3 | Basketball,Headphone,T-
Shirt
2020-06-01 | 2 | Bible,Pencil
2020-06-02 | 1 | Mask
```

REFERENCE SOLUTION

```
SELECT sell_date,
       COUNT(DISTINCT product)                                    AS num_sold,
       GROUP_CONCAT(DISTINCT product ORDER BY product SEPARATOR ',')  AS products
FROM activities
GROUP BY sell_date
ORDER BY sell_date;
```

EXERCISE 2

Team rosters

Build one line per team from `players`. A player's position may be unknown (NULL).

team	name	position
red	Asha	GK
red	Ben	FW
blue	Cara	DF
red	Dev	FW
blue	Eli	FW

Return `team`, `roster` — every player as `name (position)`, with `?` for an unknown position, ordered by name and separated by `; ` — and `positions` — the team's different known positions in alphabetical order separated by `/` (NULL when none is known). Order by `team`.

```
team | roster | positions
blue | Cara (DF); Eli (FW) | DF/FW
red | Asha (GK); Ben (FW); Dev (FW) | FW/GK
```

INPUT 1

```
CREATE TABLE players (team VARCHAR(10),
name VARCHAR(20), position VARCHAR(5));
INSERT INTO players VALUES ('red',
'Asha', 'GK'), ('red', 'Ben', 'FW'),
('blue', 'Cara', 'DF'), ('red', 'Dev',
'FW'), ('blue', 'Eli', 'FW');
```

OUTPUT 1

```
team | roster | positions
blue | Cara (DF); Eli (FW) | DF/FW
red | Asha (GK); Ben (FW); Dev (FW) |
FW/GK
```

REFERENCE SOLUTION

```
SELECT team,
       GROUP_CONCAT(CONCAT(name, ' (', COALESCE(position, '?'), ')') ORDER BY name S
       GROUP_CONCAT(DISTINCT position ORDER BY position SEPARATOR '/')
FROM players
GROUP BY team
ORDER BY team;
```

QUIZ

1. What separator does `GROUP_CONCAT(name)` use by default?

A. `', '` (comma and space)

B. `','` (comma, no space)

C. `' '` (space)

D. None — a separator is required

2. Which call is written correctly?

A. `GROUP_CONCAT(name SEPARATOR ', ' ORDER BY name)`

B. `GROUP_CONCAT(DISTINCT name ORDER BY name SEPARATOR ', ')`

C. `GROUP_CONCAT(ORDER BY name, DISTINCT name)`

D. `GROUP_CONCAT(name, ', ') ORDER BY name`

3. A report's `GROUP_CONCAT` of order ids stops after about 200 ids with no error. Why?

A. `GROUP_CONCAT` accepts at most 200 values

B. The result is cut at `group_concat_max_len`, 1,024 characters by default, with only a warning

C. The ids contain NULLs

D. The query needs `DISTINCT`

4. What is PostgreSQL's equivalent of `GROUP_CONCAT(name ORDER BY name SEPARATOR ', ')`?

A. `CONCAT_WS(', ', name)`

B. `STRING_AGG(name, ', ' ORDER BY name)`

C. `LISTAGG(name)`

D. `ARRAY(name)`

ANSWER KEY · GROUP_CONCAT — A GROUP'S VALUES AS ONE STRING

1. B MySQL's default is a bare comma. `STRING_AGG` in PostgreSQL and SQL Server is the one that has no default and requires the separator argument.

2. B The order inside the call is `DISTINCT`, the expression, `ORDER BY`, then `SEPARATOR` last. An `ORDER BY` outside the call sorts the groups, not the values in each string.

3. B MySQL silently truncates the result to `group_concat_max_len` characters. Raise it with `SET SESSION group_concat_max_len = …`, or return the ids as rows.

4. B PostgreSQL uses `STRING_AGG` with the order inside the call. `CONCAT_WS` joins arguments within a row, not rows of a group; `LISTAGG` is Oracle's name.

LESSON 6 · 14 MIN

Grouping pitfalls — ONLY_FULL_GROUP_BY, COUNT and averages

`GROUP BY` is short to write and easy to get subtly wrong. A query can run without an error, return a plausible number, and still answer a different question from the one asked — the total that counts NULLs it should not, the average that weights a tiny group like a large one, the column whose value was picked at random. This lesson collects the mistakes that show up again and again in code review and in interviews, and the rule that MySQL itself enforces to stop the worst of them.

ONLY_FULL_GROUP_BY and error 1055

Consider this query:

```
SELECT department, name, MAX(salary)
FROM employees
GROUP BY department;
```

Each output row is a department. `MAX(salary)` has one value per department, but `name` has many — which name should appear? Many people expect "the name of the highest-paid employee". SQL makes no such promise.

Older MySQL versions accepted the query and returned the name from **some** row of the group — whichever the engine met first. The result looked right on small test data and was wrong in production. Since MySQL 5.7.5 the `ONLY_FULL_GROUP_BY` SQL mode is on by default, and the query fails:

```
ERROR 1055 (42000): Expression #2 of SELECT list is not in GROUP BY clause and
contains nonaggregated column 'shop.employees.name' which is not functionally
dependent on columns in GROUP BY clause; this is incompatible with sql_mode=or
```

PostgreSQL, SQL Server and Oracle have always rejected it. The rule is the one from the GROUP BY lesson: every selected column must be in `GROUP BY`, inside an aggregate, or **functionally dependent** on the grouping columns.

Fixing it — by deciding what you meant

There are four legitimate fixes, each answering a different intention:

```
-- 1. The column identifies the group: group by it too.
SELECT department, name, MAX(salary) FROM employees GROUP BY department, name;

-- 2. You want one value from many: say which, with an aggregate.
SELECT department, MIN(name) AS first_name_alphabetically, MAX(salary) FROM er

-- 3. Every row of the group has the same value: ANY_VALUE tells MySQL so.
SELECT department_id, ANY_VALUE(department_name) AS department_name, COUNT(*)
FROM employees GROUP BY department_id;

-- 4. You wanted "the row with the max": that is not a GROUP BY question.
--    It needs a subquery or a window function (Modules 8 and 10).
```

Functional dependence is MySQL being clever: if you group by a table's primary key, every other column of that table is determined by it, so MySQL accepts `SELECT id, name, email, COUNT(*) … GROUP BY id`. PostgreSQL does the same. `ANY_VALUE` covers the cases MySQL cannot prove — it is an explicit "I know these are all equal".

What not to do: turn the mode off (`SET sql_mode = …` without `ONLY_FULL_GROUP_BY`). The error goes away and the arbitrary values come back. If an interviewer asks why the query fails, the expected answer is "the column is ambiguous within the group", followed by one of the fixes above.

COUNT(*), COUNT(col), COUNT(DISTINCT col)

Three counts, three questions:

```
SELECT department,
       COUNT(*)                    AS employees,        -- rows
       COUNT(manager_id)           AS with_manager,     -- rows where manager_
       COUNT(DISTINCT manager_id)  AS managers          -- different managers
FROM employees
GROUP BY department
ORDER BY department;
```

A report showing "employees per department" with `COUNT(manager_id)` silently drops the department head who has no manager. A report of "customers who ordered" with `COUNT(*)` on the orders table counts orders, not customers — that one is `COUNT(DISTINCT customer_id)`. Before writing a count, say the noun out loud: *rows*, *rows that have a value*, or *different values*.

WHERE versus HAVING, once more

Both filter, at different moments, and swapping them changes the answer:

```sql
-- Customers with at least one order over 100:
SELECT customer_id FROM orders WHERE amount > 100 GROUP BY customer_id;

-- Customers whose total is over 100:
SELECT customer_id FROM orders GROUP BY customer_id HAVING SUM(amount) > 100;

-- Customers whose every order is over 100:
SELECT customer_id FROM orders GROUP BY customer_id HAVING MIN(amount) > 100;
```

Three similar-sounding questions, three different queries. Also remember that `WHERE` removes rows before counting, so `WHERE status = 'paid' … COUNT(*)` counts paid orders only, and customers with no paid order vanish. If they should appear with a 0, move the condition into a conditional aggregate.

The average of averages

A chain has two stores:

store	sales	revenue
A	2	200 (average sale 100)
B	198	9900 (average sale 50)

The average sale across the chain is total revenue over total sales: 10100 / 200 = 50.5. The average of the two store averages is (100 + 50) / 2 = 75 — wrong, because it gives store A's two sales the same weight as store B's 198.

```sql
-- Wrong: averages per store, then averaged again in a report or a subquery.
-- Right: aggregate the totals.
SELECT SUM(revenue) / SUM(sales) AS avg_sale FROM store_totals;
```

The same mistake appears as an average of percentages and an average of ratios. A *weighted* figure — average price per unit across lines with different quantities — is `SUM(price * qty) / SUM(qty)`, not `AVG(price)`. When combining groups, carry **sums and counts**, never averages, and divide at the end.

NULL sums and the missing zero

`SUM` over no rows, or over only NULLs, is NULL. In a grouped query that happens when a conditional sum has no matching rows and no `ELSE 0`, or when the summed column is NULL throughout a group:

```sql
SELECT team, COALESCE(SUM(bonus), 0) AS bonus_total
FROM staff
GROUP BY team
ORDER BY team;
```

Arithmetic with that NULL then turns everything it touches into NULL — `salary_total + bonus_total` is NULL for a team without bonuses. `COALESCE` the aggregate, not just the column.

Counting after a join

A preview of the next module, because it is the most expensive grouping bug in practice. Joining orders to their order lines repeats each order once per line; summing `orders.shipping_fee` after that join counts the fee several times, and `COUNT(*)` counts lines, not orders. The fixes are to aggregate each table **before** joining, or to count `DISTINCT` keys. When a total looks too large, check whether a join has multiplied the rows.

The checklist

Before trusting a grouped result, ask:

1. Is every selected column grouped, aggregated or dependent on the key?
2. Does each count count the right noun?
3. Should this condition run before grouping (`WHERE`) or after (`HAVING`)?
4. Am I averaging averages, or carrying sums and counts?
5. Can a group's aggregate be NULL, and should it read 0?
6. Is there an `ORDER BY` with a tie-breaker?

Pitfalls

- Disabling `ONLY_FULL_GROUP_BY` instead of fixing the query.
- Expecting a non-grouped column to come from the row with the `MAX` .
- `COUNT(col)` where `COUNT(*)` was meant, and the reverse.
- Averages of averages and of percentages.
- Uncoalesced NULL aggregates poisoning later arithmetic.

Key takeaways

- Error 1055 means a selected column is ambiguous within a group — group it, aggregate it, or use `ANY_VALUE` .
- Grouping by a primary key makes the table's other columns legal to select.
- `COUNT(*)` , `COUNT(col)` and `COUNT(DISTINCT col)` answer different questions.
- Combine groups with sums and counts, then divide.
- `COALESCE(SUM(x), 0)` where a missing total should read zero.

EXERCISE 1

Three counts per department

`manager_id` is NULL for an employee with no manager.

id	department	manager_id
1	eng	NULL
2	eng	1
3	eng	1
4	eng	2
5	ops	NULL
6	ops	5
7	hr	NULL

Per department return `department`, `employees` (all of them), `with_manager` (those who have a manager) and `managers` (how many different people manage someone in it). Order by `department`.

```
department | employees | with_manager | managers
eng | 4 | 3 | 2
hr | 1 | 0 | 0
ops | 2 | 1 | 1
```

INPUT 1

```
CREATE TABLE employees (id INT PRIMARY KEY, department VARCHAR(10), manager_id INT);
INSERT INTO employees VALUES (1, 'eng', NULL), (2, 'eng', 1), (3, 'eng', 1), (4, 'eng', 2), (5, 'ops', NULL), (6, 'ops', 5), (7, 'hr', NULL);
```

OUTPUT 1

```
department | employees | with_manager | managers
eng | 4 | 3 | 2
hr | 1 | 0 | 0
ops | 2 | 1 | 1
```

REFERENCE SOLUTION

```sql
SELECT department,
       COUNT(*)                   AS employees,
       COUNT(manager_id)          AS with_manager,
       COUNT(DISTINCT manager_id) AS managers
FROM employees
GROUP BY department
ORDER BY department;
```

EXERCISE 2

A weighted unit price

Each line of `order_lines` sells `qty` units at `price` each. The average of the `price` column is not the average price paid per unit, because lines sell different quantities.

store	price	qty
A	10	1
A	20	9
B	5	4
B	5	6

Per store return `store`, `line_count` (lines sold), `units` (total quantity), `avg_price` (the plain average of `price`) and `unit_price` (total money over total units) — both rounded to 2 decimals, and `unit_price` NULL when the store sold no units. Order by `store`.

```
store | line_count | units | avg_price | unit_price
A | 2 | 10 | 15 | 19
B | 2 | 10 | 5 | 5
```

INPUT 1

```
CREATE TABLE order_lines (store
VARCHAR(5), price DECIMAL(10,2), qty
INT);
INSERT INTO order_lines VALUES ('A', 10,
1), ('A', 20, 9), ('B', 5, 4), ('B', 5,
6);
```

OUTPUT 1

```
store | line_count | units | avg_price |
unit_price
A | 2 | 10 | 15 | 19
B | 2 | 10 | 5 | 5
```

REFERENCE SOLUTION

```sql
SELECT store,
       COUNT(*)                                          AS line_count,
       SUM(qty)                                          AS units,
       ROUND(AVG(price), 2)                              AS avg_price,
       ROUND(SUM(price * qty) / NULLIF(SUM(qty), 0), 2)  AS unit_price
FROM order_lines
GROUP BY store
ORDER BY store;
```

QUIZ

1. MySQL 8 rejects `SELECT department, name, MAX(salary) FROM employees GROUP BY department` with error 1055. Why?

A. `MAX` cannot be used with `GROUP BY`

B. `name` has many values per department and the query does not say which one to show

C. `department` must be aggregated

D. Error 1055 means the table is empty

2. Which change makes MySQL accept `SELECT d.id, d.name, COUNT(*) … GROUP BY d.id` when `id` is the primary key of `d`?

A. None — it is already accepted

B. Wrap `name` in `ANY_VALUE`

C. Add `name` to GROUP BY

D. Turn off ONLY_FULL_GROUP_BY

3. Store A made 2 sales averaging 100; store B made 198 sales averaging 50. What is the average sale across both?

A. 75

B. 50.5

C. 50

D. 100

4. `SELECT team, SUM(salary) + SUM(bonus) AS pay FROM staff GROUP BY team` shows NULL for a team. What is the likely cause?

A. The team has no salaries

B. Every `bonus` in that team is NULL, so `SUM(bonus)` is NULL and the addition is NULL

C. `SUM` cannot be added

D. The team name is NULL

ANSWER KEY · GROUPING PITFALLS — ONLY_FULL_GROUP_BY, COUNT AND AVERAGES

1. B Under `ONLY_FULL_GROUP_BY` every selected column must be grouped, aggregated or functionally dependent on the grouping. The query does not mean "the name of the top earner" — that needs a subquery or window function.

2. A Grouping by a table's primary key determines every other column of that table, and MySQL 5.7+ recognises that functional dependence. The other options would also work but are unnecessary.

3. B Total revenue is 200 + 9,900 = 10,100 over 200 sales, so 50.5. Averaging the two averages (75) weights store A's two sales like store B's 198 — carry sums and counts, then divide.

4. B `SUM` over only NULLs is NULL, and anything plus NULL is NULL. `COALESCE(SUM(bonus), 0)` fixes it.

CHECKPOINT · 25 MIN

Checkpoint — Aggregation

This checkpoint covers the whole module: the aggregate functions and how they treat NULL and empty input; `GROUP BY` on columns and expressions; `HAVING` and how it differs from `WHERE`; conditional aggregation with `SUM(CASE …)` for counts, pivots and rates; `GROUP_CONCAT` with `DISTINCT`, `ORDER BY` and `SEPARATOR`; and the grouping mistakes — `ONLY_FULL_GROUP_BY`, the three `COUNT`s, and averages of averages.

How it works. Twelve questions and three queries. You need 70% on the questions and every query accepted to clear the module. You can retake it as often as you like; your best score counts.

Before you start, make sure you can answer these from memory:

- What do `COUNT(*)`, `COUNT(col)` and `COUNT(DISTINCT col)` each count?
- What do `SUM` and `COUNT` return when no rows match?
- What may appear in the SELECT list of a grouped query, and what does error 1055 mean?
- In what order do `WHERE`, `GROUP BY`, `HAVING`, `SELECT` and `ORDER BY` run?
- How do you find values that appear more than once in a column?
- Why does `COUNT(CASE WHEN … THEN 1 ELSE 0 END)` count every row?
- Where do `DISTINCT`, `ORDER BY` and `SEPARATOR` go in a `GROUP_CONCAT` call?
- Why is an average of averages usually wrong, and what do you compute instead?

The three queries are a monthly sales summary with a revenue floor, a transaction report with approved counts and amounts per month and country, and a daily product list with distinct counts and sorted names.

EXERCISE 1

Monthly sales with a revenue floor

Summarise `sales` per month, leaving out rows from the `test` region (a NULL region is a real sale and counts), and keep only months whose revenue is **at least 100**.

id	sold_on	amount	region
1	2024-01-04	40	north
2	2024-01-18	70	south
3	2024-01-20	500	test
4	2024-02-02	30	north
5	2024-02-10	50	north
6	2024-03-05	120	south
7	2024-03-25	15	north
8	2024-02-12	900	test

Return `month` (`2024-01`), `sales`, `revenue`, `avg_sale` (rounded to 2 decimals) and `biggest`. Order by `month`.

```
month | sales | revenue | avg_sale | biggest
2024-01 | 2 | 110 | 55 | 70
2024-03 | 2 | 135 | 67.5 | 120
```

INPUT 1

```
CREATE TABLE sales (id INT PRIMARY KEY,
sold_on DATE, amount DECIMAL(10,2),
region VARCHAR(10));
INSERT INTO sales VALUES (1, '2024-01-
04', 40, 'north'), (2, '2024-01-18', 70,
'south'), (3, '2024-01-20', 500,
'test'), (4, '2024-02-02', 30, 'north'),
  (5, '2024-02-10', 50, 'north'), (6,
'2024-03-05', 120, 'south'), (7, '2024-
03-25', 15, 'north'), (8, '2024-02-12',
900, 'test');
```

OUTPUT 1

```
month | sales | revenue | avg_sale |
biggest
2024-01 | 2 | 110 | 55 | 70
2024-03 | 2 | 135 | 67.5 | 120
```

REFERENCE SOLUTION

```sql
SELECT DATE_FORMAT(sold_on, '%Y-%m') AS month,
       COUNT(*)                      AS sales,
       SUM(amount)                   AS revenue,
       ROUND(AVG(amount), 2)         AS avg_sale,
       MAX(amount)                   AS biggest
FROM sales
WHERE region <> 'test' OR region IS NULL
GROUP BY DATE_FORMAT(sold_on, '%Y-%m')
HAVING SUM(amount) >= 100
ORDER BY month;
```

EXERCISE 2

Monthly transactions report

For each month and country, report the transactions and the approved ones.

id	country	state	amount	trans_date
121	US	approved	1000	2018-12-18
122	US	declined	2000	2018-12-19
123	US	approved	2000	2019-01-01
124	DE	approved	2000	2019-01-07

Return `month` (`2018-12`), `country`, `trans_count`, `approved_count`, `trans_total_amount` and `approved_total_amount` (0 when nothing was approved). `country` may be NULL; those rows form their own group. Order by `month`, then `country`.

```
month | country | trans_count | approved_count | trans_total_amount | approved_total
2018-12 | US | 2 | 1 | 3000 | 1000
2019-01 | DE | 1 | 1 | 2000 | 2000
2019-01 | US | 1 | 1 | 2000 | 2000
```

INPUT 1

```
CREATE TABLE transactions (id INT
PRIMARY KEY, country VARCHAR(5), state
VARCHAR(10), amount INT, trans_date
DATE);
INSERT INTO transactions VALUES (121,
'US', 'approved', 1000, '2018-12-18'),
(122, 'US', 'declined', 2000, '2018-12-
19'),
  (123, 'US', 'approved', 2000, '2019-
01-01'), (124, 'DE', 'approved', 2000,
'2019-01-07');
```

OUTPUT 1

```
month | country | trans_count |
approved_count | trans_total_amount |
approved_total_amount
2018-12 | US | 2 | 1 | 3000 | 1000
2019-01 | DE | 1 | 1 | 2000 | 2000
2019-01 | US | 1 | 1 | 2000 | 2000
```

REFERENCE SOLUTION

```sql
SELECT DATE_FORMAT(trans_date, '%Y-%m') AS month,
       country,
       COUNT(*)                                          AS trans_count,
       SUM(CASE WHEN state = 'approved' THEN 1 ELSE 0 END)      AS approved_count,
       SUM(amount)                                       AS trans_total_amou
       SUM(CASE WHEN state = 'approved' THEN amount ELSE 0 END)  AS approved_total_a
FROM transactions
GROUP BY DATE_FORMAT(trans_date, '%Y-%m'), country
ORDER BY month, country;
```

EXERCISE 3

Busy days and what sold

From `orders`, list the days on which **at least two different customers** ordered.

order_date	customer	product
2024-04-01	ana	pen
2024-04-01	raj	ink
2024-04-01	ana	ink
2024-04-02	raj	pad
2024-04-02	raj	pen
2024-04-03	li	pen
2024-04-03	ana	pad
2024-04-03	raj	pad

Return `order_date`, `customers` (different customers that day) and `products` (the different products, alphabetical, separated by `,`). Order by `order_date`.

```
order_date | customers | products
2024-04-01 | 2 | ink, pen
2024-04-03 | 3 | pad, pen
```

INPUT 1

```
CREATE TABLE orders (order_date DATE,
customer VARCHAR(10), product
VARCHAR(10));
INSERT INTO orders VALUES ('2024-04-01',
'ana', 'pen'), ('2024-04-01', 'raj',
'ink'), ('2024-04-01', 'ana', 'ink'),
('2024-04-02', 'raj', 'pad'),
  ('2024-04-02', 'raj', 'pen'), ('2024-
04-03', 'li', 'pen'), ('2024-04-03',
'ana', 'pad'), ('2024-04-03', 'raj',
'pad');
```

OUTPUT 1

```
order_date | customers | products
2024-04-01 | 2 | ink, pen
2024-04-03 | 3 | pad, pen
```

REFERENCE SOLUTION

```sql
SELECT order_date,
       COUNT(DISTINCT customer)                                    AS customers,
       GROUP_CONCAT(DISTINCT product ORDER BY product SEPARATOR ', ') AS products
FROM orders
GROUP BY order_date
HAVING COUNT(DISTINCT customer) ⩾ 2
ORDER BY order_date;
```

CHECKPOINT QUESTIONS

1. A table has 10 rows; `email` is NULL in 3 and two of the remaining addresses are the same. What are `COUNT(*)`, `COUNT(email)` and `COUNT(DISTINCT email)`?

A. 10, 10, 9

B. 10, 7, 6

C. 10, 7, 7

D. 7, 7, 6

2. What does `SELECT MAX(amount) FROM orders` return on an empty table?

A. 0

B. One row holding NULL

C. No rows

D. An error

3. In which logical order are the clauses of a grouped query evaluated?

A. SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY

B. FROM, WHERE, GROUP BY, HAVING, SELECT, ORDER BY

C. FROM, GROUP BY, WHERE, HAVING, SELECT, ORDER BY

D. FROM, WHERE, HAVING, GROUP BY, SELECT, ORDER BY

4. Which query finds email addresses stored more than once?

A. `SELECT email FROM users WHERE COUNT(*) > 1 GROUP BY email`

B. `SELECT DISTINCT email FROM users`

C. `SELECT email FROM users GROUP BY email HAVING COUNT(*) > 1`

D. `SELECT email, COUNT(*) FROM users HAVING COUNT(*) > 1`

5. Why does `GROUP BY MONTH(order_date)` give wrong totals in a report covering 2023 and 2024?

A. `MONTH` returns text

B. March 2023 and March 2024 fall into the same group

C. GROUP BY cannot use functions

D. It sorts months alphabetically

6. What does `SUM(CASE WHEN status = 'paid' THEN amount END)` return for a group with no paid rows?

A. 0

B. NULL

C. The total of all amounts

D. An error

7. Which expression is the share of approved transactions in each group?

A. `AVG(CASE WHEN state = 'approved' THEN 1 ELSE 0 END)`

B. `AVG(CASE WHEN state = 'approved' THEN 1 END)`

C. `COUNT(state = 'approved')`

D. `SUM(state) / COUNT(*)`

8. Where does `SEPARATOR` go in a `GROUP_CONCAT` call?

A. First, before the expression

B. Last, after any ORDER BY

C. Outside the parentheses

D. Before DISTINCT

9. `WHERE region <> 'test'` is added to a sales query. What happens to rows whose region is NULL?

A. They are kept

B. They are dropped, because `NULL <> 'test'` is unknown, not true

C. They are grouped with 'test'

D. The query fails

10. Under `ONLY_FULL_GROUP_BY`, which is the right fix when every row of a group holds the same `department_name` but you group by `department_id` from a table where it is not the key?

A. Turn the SQL mode off

B. `ANY_VALUE(department_name)`

C. `COUNT(department_name)`

D. Remove the GROUP BY

11. Five lines sell 1, 1, 1, 1 and 96 units at prices 10, 10, 10, 10 and 2. Which gives the average price paid per unit?

A. `AVG(price)`

B. `SUM(price * qty) / SUM(qty)`

C. `SUM(price) / SUM(qty)`

D. `AVG(price * qty)`

12. A query's `COUNT(*)` per order doubled after joining `orders` to `order_lines`. Why?

A. COUNT counts NULLs twice

B. The join repeats each order once per matching line, so rows are counted per line

C. GROUP BY was evaluated twice

D. The join removes duplicates

ANSWER KEY · CHECKPOINT — AGGREGATION

1. B `COUNT(*)` counts all 10 rows, `COUNT(email)` the 7 non-NULL emails, and `COUNT(DISTINCT email)` the 6 different ones (the repeated address counts once).

2. B An aggregate without `GROUP BY` always returns one row; `MAX`, `MIN`, `SUM` and `AVG` of nothing are NULL. Only `COUNT` returns 0.

3. B Rows are read, filtered by `WHERE`, grouped, filtered as groups by `HAVING`, then the SELECT list is computed and the result sorted. That is why `WHERE` cannot use aggregates or SELECT aliases.

4. C Group by the value that should be unique and keep groups with more than one row. An aggregate in `WHERE` is an error, `DISTINCT` hides the duplicates, and without `GROUP BY` the whole table is one group.

5. B `MONTH` returns 1–12 with no year, so the same month of different years merges. Group by `DATE_FORMAT(order_date, '%Y-%m')` or by `YEAR(…), MONTH(…)`.

6. B With no `ELSE`, every row gives NULL and `SUM` of only NULLs is NULL. Add `ELSE 0` when the report should show 0.

7. A The `ELSE 0` keeps every row in the average, so it equals approved / all. Without it the average is always 1; `COUNT` of a comparison counts every non-NULL comparison, true or false.

8. B The form is `GROUP_CONCAT([DISTINCT] expr [ORDER BY …] [SEPARATOR '…'])` — the separator always comes last.

9. B A comparison with NULL is unknown, and `WHERE` keeps only true rows. Add `OR region IS NULL` (or compare `COALESCE(region, '')`) when NULL rows should stay.

10. B `ANY_VALUE` tells MySQL that any value of the group will do — correct when they are known to be equal. Turning the mode off brings back arbitrary values silently; `COUNT` changes the meaning; removing `GROUP BY` collapses everything into one row.

11. B Total money (40 + 192 = 232) over total units (100) is 2.32. `AVG(price)` is 8.4 because it gives each line the same weight regardless of quantity; the other two mix the wrong totals.

12. B A join multiplies rows: an order with two lines appears twice. Count `DISTINCT` order ids, or aggregate each table before joining.
