Module 2 of 16

Selecting and filtering

aliases for columns and tables and where they work; calculated columns with arithmetic, decimal division, DIV, MOD, ROUND and CONCAT; WHERE with comparisons on numbers, text and dates; AND, OR and NOT with precedence and three-valued logic; IN, BETWEEN and LIKE; and DISTINCT with COUNT(DISTINCT).

A query earns its keep by returning exactly the rows and columns a question needs, named so the reader understands them. This module covers the two halves of that. On the column side: aliases that name what a query returns, and expressions — arithmetic, rounding, text built with `CONCAT` — that compute new columns from stored ones. On the row side: the `WHERE` clause and every tool it uses, from a single comparison to conditions combined with `AND`, `OR` and `NOT`, lists, ranges and text patterns, and finally `DISTINCT` to collapse repeated rows. Along the way it meets the two habits that separate a correct query from a nearly correct one: parenthesising mixed `AND` / `OR`, and asking what happens to rows whose value is NULL.

Columns and aliases covers `AS`, optional `AS` and the missing-comma trap, quoted aliases, table aliases and qualified names, and the clauses where an alias is visible. Expressions in SELECT covers arithmetic and precedence, MySQL's decimal `/` beside `DIV` and `MOD` (and how PostgreSQL and SQL Server differ), `ROUND` / `FLOOR` / `CEIL`, `CONCAT`, comparisons as 1/0 values, and NULL in expressions. The WHERE clause covers the comparison operators on numbers, text and dates, half-open date ranges, `IS NULL`, why `<>` drops NULL rows, and sargable filters. AND, OR and NOT covers precedence and parentheses, De Morgan's laws and three-valued logic. IN, BETWEEN and LIKE covers lists and the `NOT IN` NULL trap, inclusive ranges, wildcards, escaping and case. DISTINCT covers whole-row duplicates, NULLs, `COUNT(DISTINCT)`, `ORDER BY` after `DISTINCT`, and `GROUP BY`.

The exercises rename columns and quote an alias with a space, compute line totals with GST and split durations with `DIV` and `MOD`, filter salaries and dates, fix a precedence trap and include NULL departments on purpose, combine `IN` with `BETWEEN`, match e-mail addresses with `LIKE`, and list and count distinct values. The checkpoint asks for a discounted price list, an order review filter that needs parentheses, a range and a NULL check, and the distinct departments matched by a pattern.

CONTENTS

1	Columns and aliases — naming what a query returns	13 min
2	Expressions in SELECT — arithmetic, text and computed columns	14 min
3	The WHERE clause — keeping the rows you want	14 min
4	AND, OR and NOT — combining conditions	14 min
5	IN, BETWEEN and LIKE — lists, ranges and patterns	15 min
6	DISTINCT — removing duplicate rows	13 min
✓	Checkpoint — Selecting and filtering	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

Columns and aliases — naming what a query returns

A query's result is a table, and its columns need names. For a plain column the name comes from the table; for anything computed it is whatever the database derives from the expression — often the expression's text, which is ugly and differs between databases. An **alias** gives a column (or a table) the name you choose for the length of one query. Aliases make results readable, make computed columns addressable, and are essential once a query reads more than one table.

Column aliases with AS

```sql
SELECT first_name AS name,
       salary * 12 AS annual_salary
FROM employees
ORDER BY emp_id;
```

name	annual_salary
Asha	720000
Ravi	540000

`AS` renames the result column. The table still has `first_name`; only this result calls it `name`. Without the alias the second column's heading would be the expression itself — `salary * 12` in MySQL — which is awkward to read and harder for application code to refer to.

Rules worth knowing:

- An alias names the **result** column; it is not stored anywhere.
- Every computed column deserves one: the client code that reads the result refers to columns by name.
- Two result columns may end up with the same name (`SELECT a.id, b.id`). SQL allows it, but most client libraries then lose one of them — alias them apart.

AS is optional — use it anyway

```
SELECT first_name name, salary * 12 annual_salary FROM employees;
```

The keyword is optional in MySQL, PostgreSQL, SQLite and SQL Server. Writing it is the habit for one reason you met in the first module: a missing comma turns a column into an alias silently.

```
SELECT first_name last_name FROM employees;   -- one column, called last_name,
```

With `AS` everywhere, a bare word after a column stands out as the mistake it is.

Aliases that need quoting

An alias is an identifier, so the identifier rules apply. A simple lower-case name with underscores needs nothing. A name with a space, a capital you want kept, or a keyword needs quoting:

```
-- MySQL
SELECT salary AS `Monthly Pay`, dept AS `group` FROM employees;
-- MySQL also accepts quotes around an alias
SELECT salary AS 'Monthly Pay' FROM employees;
-- Standard SQL, PostgreSQL
SELECT salary AS "Monthly Pay" FROM employees;
```

MySQL is unusually lenient here: in the alias position it accepts backticks, single quotes and double quotes. Elsewhere in a query single quotes always make a **string**, so `ORDER BY 'Monthly Pay'` sorts by a constant string — that is, not at all. Refer to a quoted alias with backticks in MySQL. The simplest rule: choose aliases that need no quotes — `monthly_pay` — and let the application format headings for humans.

Table aliases

A table can have an alias too, with the same `AS`:

```sql
SELECT e.first_name, e.salary
FROM employees AS e
ORDER BY e.emp_id;
```

`e.first_name` is a **qualified** column name: table (or alias), a dot, the column. With one table it is optional. Its value shows when a query names two tables that both have, say, a `name` column:

```sql
SELECT e.name AS employee, d.name AS department
FROM employees AS e
JOIN departments AS d ON d.dept_id = e.dept_id;
```

Without the prefixes, `name` would be **ambiguous** and MySQL would refuse the query (`Column 'name' in field list is ambiguous`). Joins are a module of their own; the habit to form now is short table aliases (`e`, `d`, `o` for orders) and qualified names whenever more than one table is in play. Once a table has an alias, use it: in MySQL and most databases the original name is no longer visible in that query, so `employees.salary` after `FROM employees AS e` is an error.

Oracle does not accept `AS` before a table alias (`FROM employees e`); writing table aliases without `AS` is therefore the most portable form, and common in real code.

Where an alias can be used

Recall the logical order of evaluation: `FROM`, `WHERE`, `GROUP BY`, `HAVING`, `SELECT`, `ORDER BY`, `LIMIT`.

- **Table aliases** are created in `FROM`, the first step, so every clause can use them.
- **Column aliases** are created in `SELECT`. `ORDER BY` can use them in every database. `WHERE` never can. MySQL also allows them in `GROUP BY` and `HAVING`, which is convenient and not portable.

```sql
SELECT first_name, salary * 12 AS annual_salary
FROM employees
WHERE salary * 12 > 600000      -- the expression, not the alias
ORDER BY annual_salary DESC;    -- the alias is fine here
```

One more subtlety: an alias may shadow a real column name. `SELECT salary * 12 AS salary … ORDER BY salary` — which salary does `ORDER BY` mean? MySQL picks the alias. Avoid the question by never reusing a column's name for something different.

Column names from SELECT *

`SELECT *` returns the table's own column names, in the table's order. You can mix `*` with other columns — `SELECT *, salary * 12 AS annual_salary` — which MySQL allows only when `*` comes first, unless it is qualified: `SELECT salary * 12 AS annual, e.* FROM employees AS e`. Handy for exploring; still not for code that others depend on.

Naming conventions

Good aliases read like column names:

- lower case with underscores: `annual_salary`, `order_count`, `avg_price`;
- say what the value **is**, with its unit where useful: `price_inr`, `duration_min`;
- prefix booleans with `is_` / `has_`: `is_active`;
- keep table aliases to one or two letters, consistent across a codebase.

These are the names a dashboard, an API or a CSV export will show. Interview answers that alias every computed column look professional and are easier for the interviewer to check.

Key takeaways

- `expr AS name` renames a result column for one query; nothing stored changes.
- `AS` is optional; write it, so a missing comma cannot hide.
- Quote an alias with spaces using backticks in MySQL (double quotes in the standard) — or choose one that needs no quotes.
- `FROM table AS t` gives a table alias; qualify columns as `t.column`, and qualify always when several tables are involved.
- Column aliases work in `ORDER BY` everywhere, never in `WHERE`; MySQL also accepts them in `GROUP BY` / `HAVING`.

EXERCISE 1

Rename the columns

The table `employees`:

emp_id	first_name	dept	monthly_salary
2	Ravi	Support	45000
1	Asha	Sales	60000
3	Meera	NULL	52000

Return three columns with these names: `id` (the `emp_id`), `name` (the `first_name`) and `pay` (the `monthly_salary`), ordered by `emp_id`.

Output: `id | name | pay`.

```
id | name | pay
1 | Asha | 60000
2 | Ravi | 45000
3 | Meera | 52000
```

INPUT 1

```
CREATE TABLE employees (
  emp_id         INT PRIMARY KEY,
  first_name     VARCHAR(40) NOT NULL,
  dept           VARCHAR(20),
  monthly_salary INT
);
INSERT INTO employees VALUES
  (2, 'Ravi', 'Support', 45000),
  (1, 'Asha', 'Sales', 60000),
  (3, 'Meera', NULL, 52000);
```

OUTPUT 1

```
id | name | pay
1 | Asha | 60000
2 | Ravi | 45000
3 | Meera | 52000
```

REFERENCE SOLUTION

```
SELECT emp_id         AS id,
       first_name     AS name,
       monthly_salary AS pay
FROM employees
ORDER BY emp_id;
```

EXERCISE 2

An alias with a space

The table `products`:

product_id	name	price
1	Keyboard	1500
2	Monitor	9000
3	Mouse	600

Return two columns headed exactly `Product` and `Unit Price` (capitals and the space included — quote the alias), most expensive first, ties by `product_id`. Careful: in `ORDER BY`, a name in single quotes is a constant string, not your column.

Output: `Product | Unit Price`.

```
Product | Unit Price
Monitor | 9000
Keyboard | 1500
Mouse | 600
```

INPUT 1

```
CREATE TABLE products (
  product_id INT PRIMARY KEY,
  name       VARCHAR(60) NOT NULL,
  price      INT NOT NULL
);
INSERT INTO products VALUES
  (1, 'Keyboard', 1500),
  (2, 'Monitor', 9000),
  (3, 'Mouse', 600);
```

OUTPUT 1

```
Product | Unit Price
Monitor | 9000
Keyboard | 1500
Mouse | 600
```

REFERENCE SOLUTION

```
SELECT name  AS Product,
       price AS 'Unit Price'
FROM products
ORDER BY price DESC, product_id;
```

QUIZ

1. What does `SELECT first_name last_name FROM employees;` return?

A. Two columns

B. An error: missing comma

C. One column named `last_name` holding first names

D. One column named `first_name` holding last names

2. After `FROM employees AS e`, which column reference is valid in MySQL?

A. `employees.salary`

B. `e.salary`

C. `AS.salary`

D. `e→salary`

3. In which clause can a column alias **never** be used?

A. ORDER BY

B. WHERE

C. HAVING in MySQL

D. GROUP BY in MySQL

4. Why is `ORDER BY 'Unit Price'` a bug in MySQL?

A. Aliases cannot be ordered by

B. Single quotes make it a string literal, so every row sorts by the same constant

C. ORDER BY needs DESC

D. It sorts alphabetically instead of numerically

5. Why must columns be qualified, as in `e.name` and `d.name`, when two joined tables both have `name`?

A. For speed

B. Otherwise `name` is ambiguous and the query is refused

C. Qualification is required in every query

D. To rename the column

ANSWER KEY · COLUMNS AND ALIASES — NAMING WHAT A QUERY RETURNS

1. C With `AS` optional, a name after a column is an alias. The query runs and mislabels the data — which is why writing `AS` every time makes the slip visible.

2. B Once a table has an alias, the query refers to it by the alias; the original name is no longer visible in that query.

3. B WHERE is evaluated before SELECT creates the alias. ORDER BY can use it everywhere; MySQL also accepts aliases in GROUP BY and HAVING.

4. B Outside the alias position, single quotes always mean a string. Sorting by a constant leaves the order unspecified. Use backticks — `ORDER BY \`Unit Price\`` — or the underlying column.

5. B MySQL reports *Column 'name' in field list is ambiguous*. The prefix says which table's column is meant; an alias then gives the two result columns different names.

LESSON 2 · 14 MIN

Expressions in SELECT — arithmetic, text and computed columns

A select list is not limited to stored columns. Any **expression** — arithmetic on columns, a function call, a piece of text built from several columns — can stand there, and the database evaluates it once for every row. Calculated columns are how a query turns raw data into the numbers a report needs: line totals, prices with tax, ages, durations, labels. This lesson covers the arithmetic operators and their precedence, MySQL's division rules (an interview favourite), rounding, building text with `CONCAT`, and the one rule that catches everyone: anything combined with NULL is NULL.

Arithmetic on columns

```sql
SELECT name,
       price,
       qty,
       price * qty AS line_total,
       price * qty - discount AS net
FROM order_items
ORDER BY item_id;
```

For each row, the database substitutes that row's values and computes. The operators are the familiar ones: `+`, `-`, `*`, `/`, plus `DIV` and `%` / `MOD` below. Unary minus works too: `-price`.

Precedence follows mathematics — `*`, `/`, `DIV` and `%` bind tighter than `+` and `-`, and operators of equal rank go left to right:

```sql
SELECT 2 + 3 * 4   AS a,    -- 14
       (2 + 3) * 4 AS b,    -- 20
       10 - 4 - 3  AS c;    -- 3, not 9
```

When in doubt, parenthesise. A reader should not need to remember the table.

Division: / versus DIV

This is where dialects disagree, and where interview questions like to probe.

```sql
SELECT 7 / 2     AS decimal_div,   -- MySQL: 3.5000
       7 DIV 2   AS integer_div,   -- 3
       -7 DIV 2  AS neg_div,       -- -3 (truncates toward zero)
       7 % 2     AS remainder,     -- 1
       MOD(7, 2) AS remainder2;    -- 1
```

- In **MySQL**, `/` always produces a decimal result, even for two integers. By default it shows four extra digits of scale, which is why `7 / 2` prints as `3.5000`.
- `DIV` is MySQL's **integer division**: it discards the fraction.
- In **PostgreSQL, SQLite and SQL Server**, `/` between two integers is integer division: `7 / 2` is `3`. To get 3.5 there you write `7.0 / 2` or cast. A query that computes a percentage as `passed / total * 100` returns 0 for every row on those systems — and the right answer on MySQL.
- Division by zero returns **NULL** in MySQL (with a warning), and raises an error in PostgreSQL and SQL Server.

Remainders: `MOD(a, b)`, `a % b` and `a MOD b` are the same in MySQL. Together with `DIV` they split a quantity into units:

```sql
SELECT duration_min,
       duration_min DIV 60 AS hours,
       duration_min MOD 60 AS minutes
FROM calls;
```

duration_min	hours	minutes
135	2	15
59	0	59

Rounding

Money and averages need a fixed number of decimal places:

```sql
SELECT ROUND(1234.5678, 2) AS r2,     -- 1234.57
       ROUND(1234.5678)    AS r0,     -- 1235
       ROUND(1234.5678, -2) AS r_2,   -- 1200
       FLOOR(4.7) AS f,               -- 4
       CEIL(4.2)  AS c,               -- 5
       TRUNCATE(4.78, 1) AS t;        -- 4.7
```

`ROUND(x, d)` rounds to `d` places; a negative `d` rounds to tens, hundreds. For exact values (integers and `DECIMAL` columns) MySQL rounds halves away from zero: `ROUND(2.5)` is 3. `FLOOR` goes down, `CEIL` (or `CEILING`) up, and `TRUNCATE` cuts digits off without rounding.

Round in the **last** step. `ROUND(price, 2) * qty` multiplies an already-rounded price and accumulates error; `ROUND(price * qty, 2)` rounds once.

A note on types: store money as `DECIMAL(10,2)` (exact) or as integer paise/cents — never as `FLOAT`, whose binary fractions make `0.1 + 0.2` slightly more than `0.3`.

Building text

In MySQL, text is joined with `CONCAT`:

```sql
SELECT CONCAT(first_name, ' ', last_name) AS full_name,
       CONCAT(name, ' (', city, ')')      AS label,
       CONCAT('EMP-', emp_id)             AS code
FROM employees;
```

full_name	label	code
Asha Rao	Asha (Pune)	EMP-1

Numbers passed to `CONCAT` are converted to text. `CONCAT_WS(sep, a, b, …)` joins with a separator and skips NULL arguments: `CONCAT_WS(', ', city, state, country)`. Remember from the first module that `||` is **not** concatenation in MySQL. The functions module covers `UPPER`, `SUBSTRING`, `LPAD` and the rest.

Literals

A literal in the select list repeats on every row:

```sql
SELECT name, 'INR' AS currency, 18 AS gst_rate, price * 18 / 100 AS gst
FROM products;
```

Literals are useful as labels, as constant columns that a later `UNION` needs, and in expressions. Numeric literals can be integers (`18`), decimals (`0.18`) or scientific (`1.8e-1` , a floating-point value); text literals take single quotes.

Comparisons are expressions too

A comparison produces a value. In MySQL it is `1` (true), `0` (false) or `NULL` (unknown):

```sql
SELECT name, price, price > 1000 AS is_premium FROM products;
```

name	price	is_premium
Monitor	9000	1
Mouse	600	0

That is how MySQL represents booleans — `TRUE` is just `1` . PostgreSQL would print `t` / `f` ; SQL Server does not allow a bare comparison in the select list at all (you write `CASE WHEN price > 1000 THEN 1 ELSE 0 END`). The same comparisons are what `WHERE` uses to filter, in the next lesson.

NULL poisons expressions

`NULL` means "no value / unknown". Any arithmetic or concatenation that involves it returns `NULL` :

```sql
SELECT 100 + NULL      AS a,   -- NULL
       NULL * 0        AS b,   -- NULL (not 0)
       CONCAT('x', NULL) AS c; -- NULL
```

So a row whose `discount` is NULL has a NULL `price - discount` . When a missing number should count as zero, substitute it:

```sql
SELECT name, price - COALESCE(discount, 0) AS net FROM products;
```

`COALESCE(a, b, …)` returns its first non-NULL argument (`IFNULL(a, b)` is MySQL's two-argument version). NULLs get a whole module later; for now, recognise the symptom — a computed column that is unexpectedly NULL — and the cure.

Expressions everywhere

Expressions are not special to `SELECT`. The same syntax works in `WHERE` (`WHERE price * qty > 100`), `ORDER BY` (`ORDER BY price * qty DESC`), `GROUP BY` and inside aggregates (`SUM(price * qty)`). Only the alias is clause-specific.

Key takeaways

- Any expression can be a result column; give it an alias. It is computed per row and stores nothing.
- `*`, `/`, `DIV`, `%` bind tighter than `+`, `-`; parenthesise for clarity.
- MySQL's `/` is **decimal** division (`7 / 2` = 3.5); `DIV` is integer division; elsewhere `/` on integers truncates. Division by zero is NULL in MySQL.
- `ROUND(x, d)`, `FLOOR`, `CEIL`, `TRUNCATE`; round in the last step; money in `DECIMAL`.
- Join text with `CONCAT` / `CONCAT_WS`; comparisons yield 1, 0 or NULL.
- Anything combined with NULL is NULL — `COALESCE(col, 0)` substitutes a value.

EXERCISE 1

Line totals with GST

The table `order_items`:

item_id	name	price	qty
1	pen	25	4
2	notebook	199	3
3	bag	1250	1

Return `name`, the line `subtotal` (`price * qty`), and `with_gst`: the subtotal plus 18% tax, rounded to 2 decimal places. Order by `item_id`.

Output: `name | subtotal | with_gst`. (Trailing zeros after the decimal point are not printed.)

```
name | subtotal | with_gst
pen | 100 | 118
notebook | 597 | 704.46
bag | 1250 | 1475
```

INPUT 1

```
CREATE TABLE order_items (
  item_id INT PRIMARY KEY,
  name    VARCHAR(60) NOT NULL,
  price   INT NOT NULL,
  qty     INT NOT NULL
);
INSERT INTO order_items VALUES
  (1, 'pen', 25, 4),
  (2, 'notebook', 199, 3),
  (3, 'bag', 1250, 1);
```

OUTPUT 1

```
name | subtotal | with_gst
pen | 100 | 118
notebook | 597 | 704.46
bag | 1250 | 1475
```

REFERENCE SOLUTION

```sql
SELECT name,
       price * qty                AS subtotal,
       ROUND(price * qty * 1.18, 2) AS with_gst
FROM order_items
ORDER BY item_id;
```

EXERCISE 2

Hours and minutes

The table `calls` holds call lengths in minutes (NULL when the length was not recorded):

call_id	caller	duration_min
1	Asha	135
2	Ravi	59
3	Meera	60

Return `caller`, `hours` (whole hours, integer division), `minutes` (the minutes left over) and `label`: the text `<hours>h <minutes>m`. Order by `call_id`. A NULL duration gives NULL in all three computed columns.

Output: `caller | hours | minutes | label`.

```
caller | hours | minutes | label
Asha | 2 | 15 | 2h 15m
Ravi | 0 | 59 | 0h 59m
Meera | 1 | 0 | 1h 0m
```

INPUT 1

```
CREATE TABLE calls (
  call_id      INT PRIMARY KEY,
  caller       VARCHAR(40) NOT NULL,
  duration_min INT
);
INSERT INTO calls VALUES
  (1, 'Asha', 135),
  (2, 'Ravi', 59),
  (3, 'Meera', 60);
```

OUTPUT 1

```
caller | hours | minutes | label
Asha | 2 | 15 | 2h 15m
Ravi | 0 | 59 | 0h 59m
Meera | 1 | 0 | 1h 0m
```

REFERENCE SOLUTION

```sql
SELECT caller,
       duration_min DIV 60 AS hours,
       duration_min MOD 60 AS minutes,
       CONCAT(duration_min DIV 60, 'h ', duration_min MOD 60, 'm') AS label
FROM calls
ORDER BY call_id;
```

QUIZ

1. What does `SELECT 7 / 2 AS a, 7 DIV 2 AS b;` return in MySQL?

   A. 3 and 3

   B. 3.5000 and 3

   C. 3.5000 and 3.5000

   D. 3 and 3.5000

2. What is `2 + 3 * 4 - 1`?

A. 19

B. 13

C. 15

D. 20

3. A row has `price = 500` and `discount = NULL`. What is `price - discount`?

A. 500

B. 0

C. NULL

D. An error

4. Which keeps the most precision when computing a rounded line total?

A. `ROUND(price, 2) * qty`

B. `ROUND(price * qty, 2)`

C. `ROUND(price) * ROUND(qty)`

D. They are always equal

5. In MySQL, what does `SELECT 9000 > 1000 AS is_premium;` return?

A. `true`

B. `1`

C. `t`

D. An error: comparisons are only allowed in WHERE

ANSWER KEY · EXPRESSIONS IN SELECT — ARITHMETIC, TEXT AND COMPUTED COLUMNS

1. B MySQL's `/` always returns a decimal (four places of scale by default); `DIV` is integer division. In PostgreSQL, SQLite and SQL Server, `7 / 2` would be 3.

2. B Multiplication binds tighter: 3 * 4 = 12, then 2 + 12 - 1 = 13 from left to right.

3. C Any arithmetic with NULL is NULL. Use `price - COALESCE(discount, 0)` if a missing discount means none.

4. B Round once, at the end. Rounding the price first multiplies the rounding error by the quantity.

5. B MySQL's booleans are numbers: a true comparison is 1, false 0, unknown NULL. PostgreSQL would print `t`; SQL Server does not allow a bare comparison in the select list.

LESSON 3 · 14 MIN

The WHERE clause — keeping the rows you want

Most questions you ask a table are about *some* of its rows: the orders placed this month, the employees in one department, the products under a price. `WHERE` is how a query says which. It is the most-used clause after `SELECT` and `FROM`, and the one where correctness most often slips — through a comparison with NULL, a text column compared as a number, or a date range that misses its last day. This lesson covers the comparison operators, comparing numbers, text and dates, the NULL rule, and why filtering in `WHERE` is the cheapest thing a query does.

The shape

```
SELECT name, salary
FROM employees
WHERE salary ⩾ 60000
ORDER BY salary DESC, name;
```

`WHERE` comes after `FROM` and before `ORDER BY` (and `GROUP BY`, once you meet it). Logically it runs second: `FROM` produces the rows, `WHERE` tests its **condition** against each one, and only rows where the condition is **true** move on to `SELECT`. The condition is any expression that yields true, false or unknown — usually a comparison.

Comparison operators

Operator	Meaning	Example
`=`	equal	`dept = 'Sales'`
`<>` or `≠`	not equal	`status <> 'cancelled'`
`<` `>`	less / greater than	`price < 500`
`⩽` `⩾`	at most / at least	`age ⩾ 18`

Note the single `=`: SQL has no `==` (SQLite tolerates it; MySQL does not). `<>` is the standard spelling of "not equal" and `≠` an alternative every major database accepts.

Both sides can be expressions:

```
SELECT name FROM order_items WHERE price * qty > 1000;
SELECT name FROM employees WHERE salary * 12 ⩾ 1000000;
```

Column against column works too: `WHERE shipped_on > promised_on` finds late deliveries.

Comparing numbers, text and dates

Numbers compare numerically. Keep literals unquoted: `WHERE year = 2`. MySQL would convert `'2'` to a number for you, but quoting numbers hides type mistakes, and comparing a *text* column with a *number* makes MySQL convert every row's text to a number — `'12abc'` becomes 12 — and prevents an index from being used.

Text compares by the column's **collation** — its alphabet. `=` tests equality; `<` and `>` compare in dictionary order:

```
SELECT name FROM students WHERE name < 'M';    -- names before M in the alphal
```

In MySQL the default collations are **case-insensitive** and ignore trailing spaces, so `WHERE city = 'pune'` also finds `'Pune'`. PostgreSQL and SQLite compare case-sensitively. Store values consistently and don't let a query's correctness depend on the difference.

Dates compare chronologically. Write a date literal as a string in ISO format, `'YYYY-MM-DD'`, and MySQL converts it:

```
SELECT order_id, order_date
FROM orders
WHERE order_date ⩾ '2024-03-01'
ORDER BY order_date, order_id;
```

ISO strings also sort correctly *as text*, which is why the format matters: `'2024-03-01' < '2024-12-01'` is true either way, while `'01/03/2024'` versus `'12/01/2024'` would compare as text in the wrong order. Never store or compare dates in a day-first or month-first text format.

On a `DATETIME` column, `WHERE created_at = '2024-03-01'` matches only midnight. A whole day is a **half-open range**:

```
WHERE created_at ⩾ '2024-03-01' AND created_at < '2024-03-02'
```

That form also handles months of any length (`⩾ '2024-02-01' AND < '2024-03-01'`) without knowing whether February has 28 or 29 days.

NULL is never equal to anything

The rule that catches everyone: a comparison with `NULL` is neither true nor false — it is **unknown** (`NULL`), and `WHERE` keeps only rows where the condition is *true*.

```
SELECT name FROM employees WHERE manager_id = NULL;      -- always zero rows
SELECT name FROM employees WHERE manager_id IS NULL;     -- the employees with
SELECT name FROM employees WHERE manager_id IS NOT NULL;
```

`= NULL` is never true, not even for rows whose value is NULL — "is this unknown value equal to that unknown value?" has no answer. `IS NULL` and `IS NOT NULL` are the tests for presence.

The same rule hides in ordinary conditions. Given three employees, one with no department:

WHERE keeps a row only when its condition is TRUE



1. WHERE takes the rows one at a time and evaluates the condition for each. Asha's dept is Sales, so Sales <> 'Sales' is FALSE and her row is dropped.
2. Ravi's dept is Support, and Support <> 'Sales' is TRUE: his row moves on to SELECT.
3. Meera's dept is NULL. NULL <> 'Sales' is not TRUE or FALSE but UNKNOWN — nobody knows what the missing value is.
4. WHERE keeps only TRUE, and drops FALSE and UNKNOWN alike, so the answer is Ravi alone — though most people asking who is not in Sales would include Meera.
5. Saying what NULL rows should do fixes it: WHERE dept <> 'Sales' OR dept IS NULL. For Meera that is UNKNOWN OR TRUE, which is TRUE, so the answer becomes Ravi and Meera.

`WHERE dept <> 'Sales'` returns **only Ravi**. For Meera, `NULL <> 'Sales'` is unknown, so she is dropped — though most people asking "who is not in Sales?" would include her. If she belongs in the answer, say so:

```sql
WHERE dept <> 'Sales' OR dept IS NULL
```

Whenever a column can be NULL, ask what the query should do with those rows — `WHERE` will quietly drop them otherwise.

WHERE runs before SELECT

From the first module: `WHERE` cannot see select aliases, and it cannot use aggregates.

```sql
SELECT name, price * qty AS total FROM order_items WHERE total > 100;        --
SELECT name FROM products WHERE price > AVG(price);
```

Repeat the expression for the first. The second needs a subquery (`WHERE price > (SELECT AVG(price) FROM products)`), which has its own module.

Filtering early is cheap

A row removed by `WHERE` is never computed, sorted, grouped or sent over the network. Put every row condition there, and let the database do the filtering rather than fetching everything and discarding rows in application code — a habit that turns a 10-million-row transfer into a 10-row one.

`WHERE` is also where **indexes** help. With an index on `order_date` , `WHERE order_date ⩾ '2024-03-01'` jumps straight to the first matching entry instead of reading the table. One rule from the indexes module is worth knowing now: keep the column **bare** on one side. `WHERE YEAR(order_date) = 2024` makes the database compute `YEAR()` for every row and cannot use the index; `WHERE order_date ⩾ '2024-01-01' AND order_date < '2025-01-01'` can. Queries written so an index can serve them are called **sargable**.

No rows is an answer

A `WHERE` that matches nothing returns an empty result — the column headings and no rows. That is not an error: "no orders above ₹1,00,000 this month" is a valid answer, and code that reads results must handle it. The exercises print it as `(no rows)` under the heading.

Key takeaways

- `WHERE condition` keeps the rows where the condition is **true**; false and unknown are dropped.
- Operators: `=`, `<>` / `≠`, `<`, `>`, `⩽`, `⩾`; both sides can be expressions or columns.
- Text compares by collation (case-insensitive by default in MySQL); dates as `'YYYY-MM-DD'`; whole days as half-open ranges.
- A comparison with NULL is unknown: use `IS NULL` / `IS NOT NULL`, and remember `<>` silently drops NULL rows.
- `WHERE` runs before `SELECT`: no aliases, no aggregates.
- Filter in the database, early, with the column bare on one side so an index can help.

EXERCISE 1

Well-paid employees

The table `employees`:

emp_id	name	dept	salary
1	Asha	Sales	60000
2	Ravi	Support	45000
3	Meera	Eng	82000
4	Kabir	Eng	NULL

Return `name` and `salary` of employees earning **at least 60000**, highest salary first, ties by `name`. An employee whose salary is unknown is not included.

Output: `name | salary`.

```
name | salary
Meera | 82000
Asha | 60000
```

INPUT 1

```
CREATE TABLE employees (
  emp_id INT PRIMARY KEY,
  name   VARCHAR(40) NOT NULL,
  dept   VARCHAR(20),
  salary INT
);
INSERT INTO employees VALUES
  (1, 'Asha', 'Sales', 60000),
  (2, 'Ravi', 'Support', 45000),
  (3, 'Meera', 'Eng', 82000),
  (4, 'Kabir', 'Eng', NULL);
```

OUTPUT 1

```
name | salary
Meera | 82000
Asha | 60000
```

REFERENCE SOLUTION

```sql
SELECT name, salary
FROM employees
WHERE salary ⩾ 60000
ORDER BY salary DESC, name;
```

EXERCISE 2

Orders before March

The table `orders` (`order_date` is a DATE, NULL for an order not yet placed):

order_id	customer	order_date	amount
1	Asha	2024-02-14	1200
2	Ravi	2024-03-01	450
3	Meera	2024-01-30	800
4	Dev	NULL	300

Return `order_id`, `customer` and `order_date` for orders placed **before 1 March 2024**, oldest first, ties by `order_id`.

Output: `order_id | customer | order_date`.

```
order_id | customer | order_date
3 | Meera | 2024-01-30
1 | Asha | 2024-02-14
```

INPUT 1

```
CREATE TABLE orders (
  order_id   INT PRIMARY KEY,
  customer   VARCHAR(40) NOT NULL,
  order_date DATE,
  amount     INT NOT NULL
);
INSERT INTO orders VALUES
  (1, 'Asha', '2024-02-14', 1200),
  (2, 'Ravi', '2024-03-01', 450),
  (3, 'Meera', '2024-01-30', 800),
  (4, 'Dev', NULL, 300);
```

OUTPUT 1

```
order_id | customer | order_date
3 | Meera | 2024-01-30
1 | Asha | 2024-02-14
```

REFERENCE SOLUTION

```sql
SELECT order_id, customer, order_date
FROM orders
WHERE order_date < '2024-03-01'
ORDER BY order_date, order_id;
```

QUIZ

1. How many rows does `SELECT * FROM employees WHERE manager_id = NULL;` return?

A. The rows with no manager

B. Zero, always

C. Every row

D. An error

2. `dept` is `'Sales'`, `'Support'` and NULL in three rows. How many does `WHERE dept <> 'Sales'` return?

A. Two

B. One

C. Three

D. Zero

3. Which condition selects every moment of 1 March 2024 in a `DATETIME` column `created_at`?

A. `created_at = '2024-03-01'`

B. `created_at ⩾ '2024-03-01' AND created_at < '2024-03-02'`

C. `created_at ⩽ '2024-03-01'`

D. `created_at BETWEEN '2024-03-01' AND '2024-03-01'`

4. Which filter can use an index on `order_date`?

A. `WHERE YEAR(order_date) = 2024`

B. `WHERE order_date ⩾ '2024-01-01' AND order_date < '2025-01-01'`

C. Both equally

D. Neither

5. With MySQL's default collation, does `WHERE city = 'pune'` match a row whose city is `'Pune'`?

A. Yes

B. No

C. Only with LIKE

D. Only if city is indexed

ANSWER KEY · THE WHERE CLAUSE — KEEPING THE ROWS YOU WANT

1. B A comparison with NULL is unknown, never true, so WHERE keeps nothing. Use `WHERE manager_id IS NULL`.

2. B Only 'Support'. For the NULL row, `NULL <> 'Sales'` is unknown and the row is dropped. Add `OR dept IS NULL` to include it.

3. B `'2024-03-01'` means midnight, so equality and a one-day BETWEEN only match midnight. A half-open range covers the whole day.

4. B Wrapping the column in a function forces a computation on every row. Keeping the column bare and comparing with a range is sargable — the index can jump to the first match.

5. A MySQL's default collations are case-insensitive. PostgreSQL and SQLite compare case-sensitively and would not match.

LESSON 4 · 14 MIN

AND, OR and NOT — combining conditions

Real questions have more than one condition: books *or* games, under ₹500, *and* in stock, *not* discontinued. SQL combines conditions with three logical operators — `AND`, `OR` and `NOT`. They are simple on their own and dangerous together, because of **precedence** (which one binds first) and because SQL logic has a third truth value, **unknown**, that the familiar truth tables leave out. This lesson covers all three operators, the parentheses rule, rewriting negations, and the three-valued truth tables that decide what NULL rows do.

AND: both must hold

```sql
SELECT name, price, stock
FROM products
WHERE price < 500 AND stock > 0
ORDER BY name;
```

A row survives only if **every** `AND`ed condition is true. Each `AND` narrows the result. The order you write them in does not matter to the answer — `stock > 0 AND price < 500` is the same filter — and the optimizer chooses which to check first.

OR: at least one must hold

```sql
SELECT name, category
FROM products
WHERE category = 'books' OR category = 'games'
ORDER BY name;
```

A row survives if **any** `OR`ed condition is true. Each `OR` widens the result. (A list of alternatives on one column reads better as `category IN ('books', 'games')` — the next lesson.)

NOT: negate

```sql
SELECT name FROM products WHERE NOT discontinued;           -- discontinued is
SELECT name FROM products WHERE NOT (price > 1000);         -- same as price
SELECT name FROM products WHERE category NOT IN ('books');  -- NOT also combi
```

`NOT` flips true to false and false to true. In MySQL a 0/1 column can be tested directly (`WHERE discontinued`, `WHERE NOT discontinued`) because MySQL's booleans are numbers. For simple comparisons, prefer the direct operator: `price ⩽ 1000` reads better than `NOT (price > 1000)`.

Precedence: AND before OR

The precedence, tightest first: comparisons, then `NOT`, then `AND`, then `OR`. Just as `2 + 3 * 4` multiplies first, `a OR b AND c` evaluates `AND` first. Consider: *books or games, priced under 500*.

```sql
-- WRONG
SELECT name, category, price
FROM products
WHERE category = 'books' OR category = 'games' AND price < 500;
```

That is read as

```sql
WHERE category = 'books' OR (category = 'games' AND price < 500)
```

— every book, at any price, plus the cheap games. A ₹2,000 book slips through. The fix is parentheses around the `OR` group:

```sql
-- RIGHT
SELECT name, category, price
FROM products
WHERE (category = 'books' OR category = 'games') AND price < 500
ORDER BY name;
```

This is the single most common filtering bug, and it is invisible in testing whenever the data happens to have no expensive books. The rule: **whenever `AND` and `OR` appear in one condition, parenthesise every `OR` group**, even when precedence would have done the right thing — a reader should not have to work it out.

Rewriting negations: De Morgan

Negating a combination swaps `AND` and `OR`:

- `NOT (a AND b)` is `NOT a OR NOT b`
- `NOT (a OR b)` is `NOT a AND NOT b`

So "orders that are neither paid nor refunded":

```sql
WHERE NOT (status = 'paid' OR status = 'refunded')
-- equals
WHERE status <> 'paid' AND status <> 'refunded'
-- equals
WHERE status NOT IN ('paid', 'refunded')
```

A classic slip is writing `status <> 'paid' OR status <> 'refunded'`, which is true for *every* row — anything that is `'paid'` is not `'refunded'`. When a negated condition looks odd, rewrite it with De Morgan and read it aloud.

Three-valued logic

SQL conditions can be **true**, **false** or **unknown**. Unknown comes from NULL: `dept = 'Sales'` is unknown when `dept` is NULL. The operators extend to it with one principle — if the answer would be the same whatever the unknown value turned out to be, use that answer; otherwise the result is unknown.

`a`	`b`	`a AND b`	`a OR b`
true	unknown	unknown	true
false	unknown	false	unknown
unknown	unknown	unknown	unknown

and `NOT unknown` is `unknown`. `false AND anything` is false, `true OR anything` is true; the rest stays unknown. `WHERE` keeps a row only when the whole condition is **true**, so unknown behaves like false at the end — but not in the middle, and that is where the surprises live.

NOT does not rescue NULL rows

Take

emp_id	name	dept
1	Asha	Sales
2	Ravi	Support
3	Meera	NULL

- `WHERE dept = 'Sales'` → Asha.
- `WHERE NOT dept = 'Sales'` → Ravi only.

Meera is in **neither** answer. For her the comparison is unknown, and `NOT unknown` is unknown. A filter and its negation do not always add up to the whole table — whenever NULLs are possible. To put Meera with the "not Sales" group, say so:

```
WHERE dept <> 'Sales' OR dept IS NULL
```

The same thing happens with `NOT IN` and a NULL inside the list, which the next lesson shows; it is one of the most-asked SQL interview traps.

Readable compound conditions

Long conditions get one condition per line, `AND` / `OR` leading, and parentheses aligned:

```
SELECT order_id, status, amount, country
FROM orders
WHERE status = 'pending'
  AND (amount > 10000 OR country <> 'IN')
  AND created_on ⩾ '2024-01-01'
ORDER BY order_id;
```

Leading operators make it easy to comment a line out while debugging, and the parentheses show at a glance which conditions are alternatives.

Short-circuiting is not guaranteed

In C or Java, `a && b` never evaluates `b` when `a` is false. SQL makes no such promise: the optimizer may evaluate the conditions in any order. Do not write `WHERE qty <> 0 AND total / qty > 10` expecting the first condition to protect the second — in MySQL the division by zero yields NULL anyway, but in other databases it can raise an error. Make each condition safe on its own (here, `NULLIF(qty, 0)` in the divisor, which the NULLs module covers).

Key takeaways

- `AND`: every condition true. `OR`: at least one. `NOT`: flips true and false.
- Precedence: `NOT` > `AND` > `OR`. **Parenthesise every OR group** that sits next to an `AND`.
- De Morgan: `NOT (a OR b)` = `NOT a AND NOT b`; `x <> 1 OR x <> 2` is always true.
- Three-valued logic: `false AND unknown` is false, `true OR unknown` is true, everything else with unknown is unknown; `WHERE` keeps only true.
- A condition and its `NOT` both drop NULL rows — add `OR col IS NULL` when they belong.
- Don't rely on evaluation order between conditions.

EXERCISE 1

Books or games under 500

The table `products`:

product_id	name	category	price
1	SQL Primer	books	450
2	Chess Set	games	899
3	Atlas	books	2000
4	Card Game	games	299
5	Headphones	audio	350

Return `name`, `category` and `price` of products that are **books or games** and cost **less than 500**, cheapest first, ties by `product_id`.

Output: `name | category | price`.

```
name | category | price
Card Game | games | 299
SQL Primer | books | 450
```

INPUT 1

```
CREATE TABLE products (
  product_id INT PRIMARY KEY,
  name       VARCHAR(60) NOT NULL,
  category   VARCHAR(20) NOT NULL,
  price      INT NOT NULL
);
INSERT INTO products VALUES
  (1, 'SQL Primer', 'books', 450),
  (2, 'Chess Set', 'games', 899),
  (3, 'Atlas', 'books', 2000),
  (4, 'Card Game', 'games', 299),
  (5, 'Headphones', 'audio', 350);
```

OUTPUT 1

```
name | category | price
Card Game | games | 299
SQL Primer | books | 450
```

REFERENCE SOLUTION

```sql
SELECT name, category, price
FROM products
WHERE (category = 'books' OR category = 'games')
  AND price < 500
ORDER BY price, product_id;
```

EXERCISE 2

Everyone not in Sales

The table `employees` (`dept` is NULL for someone not yet assigned):

emp_id	name	dept	salary
1	Asha	Sales	60000
2	Ravi	Support	45000
3	Meera	NULL	52000

Return `name` and `dept` of every employee who is **not in Sales** — and an employee with no department counts as not in Sales. Order by `emp_id`.

Output: `name | dept`.

```
name | dept
Ravi | Support
Meera | NULL
```

INPUT 1

```
CREATE TABLE employees (
  emp_id INT PRIMARY KEY,
  name   VARCHAR(40) NOT NULL,
  dept   VARCHAR(20),
  salary INT
);
INSERT INTO employees VALUES
  (1, 'Asha', 'Sales', 60000),
  (2, 'Ravi', 'Support', 45000),
  (3, 'Meera', NULL, 52000);
```

OUTPUT 1

```
name | dept
Ravi | Support
Meera | NULL
```

REFERENCE SOLUTION

```
SELECT name, dept
FROM employees
WHERE dept <> 'Sales' OR dept IS NULL
ORDER BY emp_id;
```

QUIZ

1. How does MySQL read `WHERE a = 1 OR b = 2 AND c = 3`?

A. `(a = 1 OR b = 2) AND c = 3`

B. `a = 1 OR (b = 2 AND c = 3)`

C. Left to right, as written

D. It is a syntax error

2. Which condition is true for **every** row with a non-NULL `status`?

A. `status <> 'paid' AND status <> 'refunded'`

B. `status <> 'paid' OR status <> 'refunded'`

C. `NOT (status = 'paid' OR status = 'refunded')`

D. `status NOT IN ('paid', 'refunded')`

3. What is `FALSE AND NULL` in SQL?

A. NULL

B. FALSE

C. TRUE

D. An error

4. Rows have `dept` = 'Sales', 'Support', NULL. How many rows do `WHERE dept = 'Sales'` and `WHERE NOT dept = 'Sales'` return **together**?

A. 3

B. 2

C. 1

D. 0

5. `NOT (a OR b)` is equivalent to:

A. `NOT a OR NOT b`

B. `NOT a AND NOT b`

C. `a AND b`

D. `NOT a AND b`

ANSWER KEY · AND, OR AND NOT — COMBINING CONDITIONS

1. B AND has higher precedence than OR, as multiplication has over addition. Parentheses make the intended grouping explicit.

2. B A status cannot be both 'paid' and 'refunded', so it always differs from at least one. The other three are De Morgan rewrites of one another: neither paid nor refunded.

3. B Whatever the unknown value is, FALSE AND it is FALSE, so the result is FALSE. `TRUE AND NULL` is NULL, and `TRUE OR NULL` is TRUE.

4. B One each: 'Sales', then 'Support'. The NULL row is unknown under both, and NOT unknown is unknown — a filter and its negation do not cover a table that has NULLs.

5. B De Morgan: negating an OR gives the AND of the negations — neither a nor b.

LESSON 5 · 15 MIN

IN, BETWEEN and LIKE — lists, ranges and patterns

Three conditions come up so often that SQL gives them their own syntax. `IN` tests a value against a **list**, `BETWEEN` against a **range**, and `LIKE` against a text **pattern**. Each could be written with `=`, `>=` and `OR`, but the shorthand is shorter, clearer and, for `LIKE`, impossible to replace otherwise. Each also has a trap: `NOT IN` with a NULL, `BETWEEN` on timestamps, and wildcards that need escaping.

IN: one of a list

```sql
SELECT name, category
FROM products
WHERE category IN ('books', 'games', 'music')
ORDER BY name;
```

`x IN (a, b, c)` is exactly `x = a OR x = b OR x = c`, written once. The list can hold numbers, strings or dates — of the column's type — and can be as long as you need:

```sql
WHERE order_id IN (101, 205, 317)
WHERE status NOT IN ('cancelled', 'refunded')
```

`NOT IN` is the negation: `x <> a AND x <> b AND …`. The list can also be a **subquery** — `WHERE customer_id IN (SELECT customer_id FROM orders)` — which is the subqueries module's business, and the reason the next section matters.

The NOT IN NULL trap

Expand `NOT IN` with a NULL in the list:

```sql
WHERE dept_id NOT IN (10, NULL)
-- means
WHERE dept_id <> 10 AND dept_id <> NULL
```

`dept_id <> NULL` is unknown for every row, and `anything AND unknown` is never true. The query returns **no rows at all**, silently. Nobody writes a literal NULL in a list on purpose, but `NOT IN (SELECT manager_id FROM employees)` returns NULL whenever one employee has no

manager — and the query that worked yesterday returns nothing today. This is one of the most-asked SQL interview questions. The fixes come in the subqueries module (`NOT EXISTS`, or filtering the NULLs out); the thing to remember now is that **`NOT IN` and NULLs do not mix**.

`IN` itself is tolerant: `dept_id IN (10, NULL)` is `dept_id = 10 OR dept_id = NULL`, which is true for 10 and unknown otherwise — the NULL in the list just never matches. And a NULL on the left (`NULL IN (…)`) is unknown, so such a row is excluded by both `IN` and `NOT IN`.

BETWEEN: an inclusive range

```sql
SELECT name, price
FROM products
WHERE price BETWEEN 500 AND 1000
ORDER BY price, name;
```

`x BETWEEN low AND high` is `x ⩾ low AND x ⩽ high` — **both ends included**. A product priced exactly 500 or exactly 1000 is in. Points to remember:

- The low end comes first. `BETWEEN 1000 AND 500` is `x ⩾ 1000 AND x ⩽ 500`, which matches nothing.
- `NOT BETWEEN low AND high` is `x < low OR x > high`.
- It works on text and dates as well as numbers: `name BETWEEN 'A' AND 'M'` — though note that `'Mohan'` is *after* `'M'` and so excluded; the upper end of a text range is easy to get wrong.

Dates in `DATE` columns are fine:

```sql
WHERE order_date BETWEEN '2024-03-01' AND '2024-03-31'
```

On a `DATETIME` column, though, `'2024-03-31'` means midnight at the start of the 31st, so almost the whole last day is missed. For timestamps use the half-open range from the previous lesson: `⩾ '2024-03-01' AND < '2024-04-01'`. Many teams simply avoid `BETWEEN` for time for that reason.

LIKE: text patterns

```sql
SELECT name, email
FROM customers
WHERE email LIKE '%@gmail.com'
ORDER BY name;
```

`LIKE` compares text against a **pattern** with two wildcards:

Wildcard	Matches	Example	Matches
`%`	any run of characters, including none	`'A%'`	Asha, Arjun, A
`_`	exactly one character	`'_at'`	cat, bat — not at, not chat

Common shapes:

```sql
WHERE name LIKE 'S%'         -- starts with S
WHERE name LIKE '%a'         -- ends with a
WHERE name LIKE '%an%'       -- contains "an"
WHERE code LIKE 'CS___'      -- CS followed by exactly three characters
WHERE name LIKE '_a%'        -- second letter is a
WHERE phone NOT LIKE '+91%' -- not an Indian number
```

Without a wildcard, `LIKE 'Pune'` is just an equality test — use `=`.

Case. With MySQL's default collations `LIKE` is case-insensitive: `'a%'` matches `Asha`. PostgreSQL's `LIKE` is case-sensitive and adds `ILIKE` for the insensitive form; SQLite's is insensitive for ASCII letters.

Escaping. To find a literal `%` or `_` — a discount code like `50%OFF`, a file name with an underscore — escape it. MySQL treats backslash as the escape character by default; any database accepts an explicit one:

```sql
WHERE code LIKE '50\%%'                  -- MySQL: starts with "50%"
WHERE code LIKE '50!%%' ESCAPE '!'       -- portable
WHERE file_name LIKE '%\_v2%'            -- contains "_v2"
```

An unescaped `_` in a pattern is a common silent bug: `LIKE 'user_1%'` also matches `userA1`.

NULL. `NULL LIKE 'a%'` is unknown, so NULL values match neither `LIKE` nor `NOT LIKE`.

LIKE and indexes

A pattern with a fixed **prefix** — `'Sha%'` — can use an index on the column, because matching strings sit together in sorted order. A **leading wildcard** — `'%sha'`, `'%sha%'` — cannot: the database must test every row. For searching words inside long text, databases offer **full-text search** (MySQL's `MATCH … AGAINST` on a `FULLTEXT` index), which this course mentions but does not judge.

Putting them together

```sql
SELECT product_id, name, category, price
FROM products
WHERE category IN ('books', 'games')
  AND price BETWEEN 200 AND 800
  AND name NOT LIKE '%(used)%'
ORDER BY price DESC, product_id;
```

Each is an ordinary condition and combines with `AND`, `OR` and `NOT` under the rules of the previous lesson — `IN` and `BETWEEN` bind like comparisons, tighter than `NOT`, `AND` and `OR`.

Key takeaways

- `x IN (a, b, c)` = `x = a OR x = b OR x = c`; `NOT IN` = all `<>` joined by `AND`.
- **`NOT IN` with a NULL in the list returns no rows** — watch subqueries that can return NULL.
- `BETWEEN low AND high` includes both ends; low first; use half-open ranges for timestamps.
- `LIKE`: `%` any run (even empty), `_` exactly one character; case-insensitive by default in MySQL.
- Escape a literal `%` / `_` with `\` in MySQL or `ESCAPE '!'` anywhere.
- NULL matches none of `IN`, `NOT IN`, `LIKE`, `NOT LIKE`; leading `%` defeats an index.

EXERCISE 1

Mid-priced books, games and music

The table `products`:

product_id	name	category	price
1	Novel	books	300
2	Board Game	games	1200
3	Vinyl	music	800
4	Speaker	audio	500
5	Comic	books	150

Return `name`, `category` and `price` of products in the categories **books, games or music** whose price is **between 300 and 800 inclusive**, cheapest first, ties by `product_id`.

Output: `name | category | price`.

```
name | category | price
Novel | books | 300
Vinyl | music | 800
```

INPUT 1

```
CREATE TABLE products (
  product_id INT PRIMARY KEY,
  name       VARCHAR(60) NOT NULL,
  category   VARCHAR(20) NOT NULL,
  price      INT NOT NULL
);
INSERT INTO products VALUES
  (1, 'Novel', 'books', 300),
  (2, 'Board Game', 'games', 1200),
  (3, 'Vinyl', 'music', 800),
  (4, 'Speaker', 'audio', 500),
  (5, 'Comic', 'books', 150);
```

OUTPUT 1

```
name | category | price
Novel | books | 300
Vinyl | music | 800
```

REFERENCE SOLUTION

```
SELECT name, category, price
FROM products
WHERE category IN ('books', 'games', 'music')
  AND price BETWEEN 300 AND 800
ORDER BY price, product_id;
```

EXERCISE 2

Gmail users named A or S

The table `customers` (`email` may be NULL):

customer_id	name	email	city
1	Asha	asha.rao@gmail.com	Pune
2	Ravi	ravi@yahoo.com	Chennai
3	Sameer	sameer@gmail.com	Delhi
4	Anil	anil@notgmail.com	Pune

Return `name` and `email` of customers whose e-mail address **ends with** `@gmail.com` and whose name **starts with A or S**, ordered by `name`, ties by `customer_id`.

Output: `name | email`.

```
name | email
Asha | asha.rao@gmail.com
Sameer | sameer@gmail.com
```

INPUT 1

```
CREATE TABLE customers (
  customer_id INT PRIMARY KEY,
  name        VARCHAR(40) NOT NULL,
  email       VARCHAR(80),
  city        VARCHAR(40)
);
INSERT INTO customers VALUES
  (1, 'Asha', 'asha.rao@gmail.com',
'Pune'),
  (2, 'Ravi', 'ravi@yahoo.com',
'Chennai'),
  (3, 'Sameer', 'sameer@gmail.com',
'Delhi'),
  (4, 'Anil', 'anil@notgmail.com',
'Pune');
```

OUTPUT 1

```
name | email
Asha | asha.rao@gmail.com
Sameer | sameer@gmail.com
```

REFERENCE SOLUTION

```sql
SELECT name, email
FROM customers
WHERE email LIKE '%@gmail.com'
  AND (name LIKE 'A%' OR name LIKE 'S%')
ORDER BY name, customer_id;
```

QUIZ

1. Which values does `price BETWEEN 100 AND 200` match?

A. 101 to 199

B. 100 to 199

C. 100 to 200, both included

D. 101 to 200

2. `dept_id` holds 10, 20 and 30 in three rows. What does `WHERE dept_id NOT IN (10, NULL)` return?

A. 20 and 30

B. 30 only

C. No rows

D. All three

3. Which strings does `LIKE '_a%'` match?

A. Strings that start with a

B. Strings whose second character is a

C. Strings that contain a

D. Strings of exactly two characters ending in a

4. How do you find codes that start with the literal text `50%` in MySQL?

A. `LIKE '50%'`

B. `LIKE '50\%%'`

C. `LIKE '50%%'`

D. `= '50%'`

5. Which pattern cannot use an ordinary index on `name`?

A. `name LIKE 'Sha%'`

B. `name LIKE '%sha'`

C. `name = 'Shaan'`

D. `name ⩾ 'S'`

ANSWER KEY · IN, BETWEEN AND LIKE — LISTS, RANGES AND PATTERNS

1. C BETWEEN is inclusive at both ends: `price ⩾ 100 AND price ⩽ 200`.

2. C It expands to `dept_id <> 10 AND dept_id <> NULL`. The second comparison is unknown for every row, so the AND is never true — the classic NOT IN trap.

3. B `_` is exactly one character, then `a`, then `%` any run. So 'Ravi' and 'Sameer' match; 'Asha' does not.

4. B The escaped `\%` is a literal percent sign and the final `%` is the wildcard. `'50%'` alone matches anything starting with 50; `= '50%'` matches only the exact text.

5. B A leading wildcard means matching strings are scattered through the sorted index, so every row must be tested. A fixed prefix keeps the matches together.

LESSON 6 · 13 MIN

DISTINCT — removing duplicate rows

A query returns one result row for every input row that survives `WHERE`. Often that means repetition: list the cities your customers live in and Pune appears forty times. `DISTINCT` collapses identical result rows into one. It is simple to write and easy to misread — it applies to the **whole row**, it treats NULLs as equal, and it is sometimes used to paper over a query that produces duplicates by mistake. This lesson covers all of that, plus `COUNT(DISTINCT …)` and how `DISTINCT` relates to `GROUP BY`.

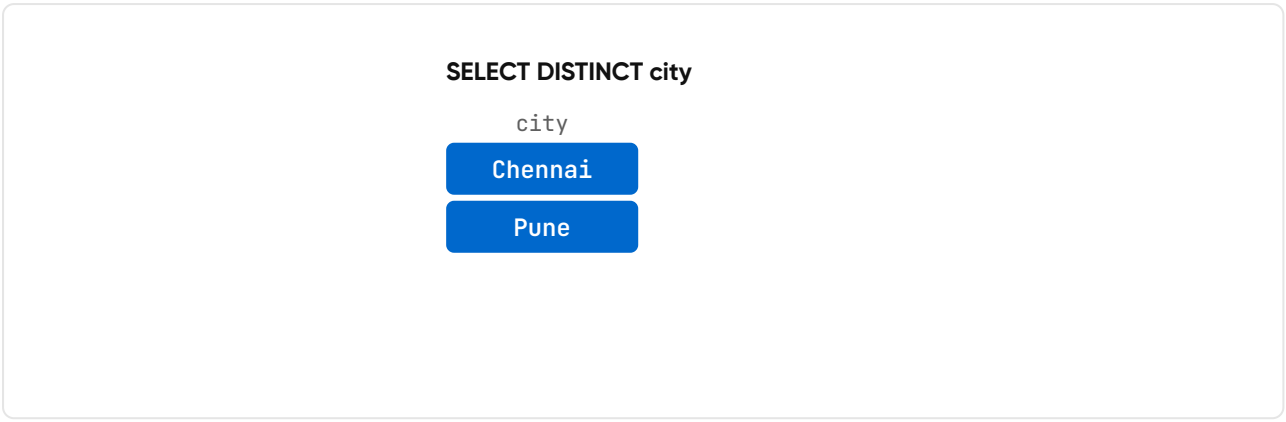
One row per value

```sql
SELECT city FROM customers ORDER BY city;
```

city
Chennai
Pune
Pune
Pune

```sql
SELECT DISTINCT city FROM customers ORDER BY city;
```

DISTINCT keeps one copy of each identical result row



1. Without DISTINCT the query returns one row per customer: 4 rows, and Pune repeats for every customer who lives there.
2. DISTINCT runs after SELECT has computed the columns and compares whole result rows. 2 rows are identical to an earlier one.
3. Each duplicate is folded into the first copy. Rows only merge when every selected column matches — add name to the select list and the Punes would all differ again.
4. 2 distinct rows remain, sorted by ORDER BY, which runs after DISTINCT. How many customers each city had is gone; COUNT with GROUP BY keeps it.

`DISTINCT` goes right after `SELECT` and removes **duplicate result rows**. In the logical order it runs after `SELECT` has computed the columns and before `ORDER BY` sorts them. The opposite, `SELECT ALL`, is the default and is never written.

DISTINCT applies to the whole row

`DISTINCT` is not a function of the first column; it compares entire result rows:

```
SELECT DISTINCT city, country FROM customers ORDER BY country, city;
```

returns each **(city, country) pair** once. Pune, India appears once however many customers live there; if another country also had a Pune, that pair would be a second row. A common mistake is writing `SELECT DISTINCT city, name` and expecting one row per city — every different name makes a different row, so the duplicates come back.

The parenthesised form is misleading:

```
SELECT DISTINCT(city), name FROM customers;
```

looks as if it applies to `city` only. It does not: the parentheses just surround the expression `city`, and `DISTINCT` still applies to `(city, name)`. Never write it that way. (PostgreSQL has

`DISTINCT ON (col)` for "one row per value of col", which MySQL lacks; there you would use `GROUP BY` or a window function, both later in the course.)

DISTINCT on expressions

`DISTINCT` compares the **computed** columns, so you can de-duplicate a derived value:

```
SELECT DISTINCT YEAR(order_date) AS order_year
FROM orders
ORDER BY order_year;
```

order_year
2023
2024

NULLs collapse into one

For `DISTINCT`, two NULLs count as the same value — the one place in SQL where NULLs are treated as equal. A column with NULL in five rows contributes a single NULL row:

```
SELECT DISTINCT dept FROM employees ORDER BY dept;
```

dept
NULL
Sales
Support

In MySQL, NULL sorts **first** in ascending order (and last in descending), which is why it heads the list. PostgreSQL and Oracle do the opposite by default — NULLs last ascending — and offer `NULLS FIRST` / `NULLS LAST` to choose; MySQL does not have that syntax.

Counting distinct values

How many different cities do customers come from? `COUNT(DISTINCT col)`:

```sql
SELECT COUNT(*)             AS customers,
       COUNT(city)          AS with_city,
       COUNT(DISTINCT city) AS cities
FROM customers;
```

customers	with_city	cities
6	5	2

Three different counts, three different questions:

- `COUNT(*)` — rows, duplicates and NULLs included;
- `COUNT(city)` — rows where `city` is not NULL;
- `COUNT(DISTINCT city)` — different non-NULL values. NULL is **not** counted, even though `SELECT DISTINCT city` shows it as a row.

That last difference — `SELECT DISTINCT` shows one NULL row, `COUNT(DISTINCT)` ignores NULL — is a favourite interview "what does this return?". MySQL also accepts several columns, `COUNT(DISTINCT city, country)`, counting distinct pairs that have no NULL in them; that form is not standard.

DISTINCT and ORDER BY

Because `DISTINCT` merges rows, `ORDER BY` after it may only use what survived — the selected columns (or expressions of them):

```sql
SELECT DISTINCT city FROM customers ORDER BY joined;   -- refused
```

Which `joined` should Pune be sorted by, when forty customers in Pune joined on forty dates? MySQL 8 refuses the query (`Expression #1 of ORDER BY clause is not in SELECT list`), as do PostgreSQL and SQL Server. Order by a selected column, or use `GROUP BY city ORDER BY MIN(joined)` to say which date you mean.

DISTINCT versus GROUP BY

These return the same rows:

```
SELECT DISTINCT city FROM customers;
SELECT city FROM customers GROUP BY city;
```

and MySQL usually runs them the same way — by sorting or hashing the values. The difference is purpose. `GROUP BY` exists to **compute something per group** (`COUNT(*)` , `SUM(amount)`); `DISTINCT` only removes duplicate output rows. Use `DISTINCT` to list unique values, `GROUP BY` to summarise; the aggregation module takes `GROUP BY` from there.

DISTINCT is not free, and not a fix

Removing duplicates means sorting or hashing the whole result — on a large result that is real work and memory. It is worth it when the question is "which different values?". It is a smell when it is added because "the query returned duplicates": usually a join matched more rows than intended, and `DISTINCT` hides the symptom while the query still does the extra work and may hide wrong totals. When you reach for `DISTINCT` to fix a result, first ask **why** the duplicates are there.

Another subtle case: `DISTINCT` is applied **before** `LIMIT` , so `SELECT DISTINCT city … ORDER BY city LIMIT 3` gives three *different* cities.

Key takeaways

- `SELECT DISTINCT` removes duplicate result rows; it runs after `SELECT` , before `ORDER BY` and `LIMIT` .
- It applies to **all** selected columns together; `DISTINCT(col)` is just `DISTINCT col` .
- NULLs collapse into one row; MySQL sorts NULL first ascending.
- `COUNT(*)` rows, `COUNT(col)` non-NULL values, `COUNT(DISTINCT col)` different non-NULL values.
- After `DISTINCT` , `ORDER BY` may only use selected columns.
- Same rows as `GROUP BY` on those columns; `GROUP BY` is for aggregates.
- Don't use `DISTINCT` to hide duplicates a join created — find the cause.

EXERCISE 1

Cities we ship to

The table `customers` (`city` may be NULL):

customer_id	name	email	city
1	Asha	NULL	Pune
2	Ravi	NULL	Chennai
3	Meera	NULL	Pune
4	Dev	NULL	Pune

Return each different `city` once, in alphabetical order. A missing city appears once, as `NULL` — and in MySQL, NULL sorts first.

Output: one column `city`.

```
city
Chennai
Pune
```

INPUT 1

```
CREATE TABLE customers (
  customer_id INT PRIMARY KEY,
  name        VARCHAR(40) NOT NULL,
  email       VARCHAR(80),
  city        VARCHAR(40)
);
INSERT INTO customers VALUES
  (1, 'Asha', NULL, 'Pune'),
  (2, 'Ravi', NULL, 'Chennai'),
  (3, 'Meera', NULL, 'Pune'),
  (4, 'Dev', NULL, 'Pune');
```

OUTPUT 1

```
city
Chennai
Pune
```

REFERENCE SOLUTION

```
SELECT DISTINCT city
FROM customers
ORDER BY city;
```

EXERCISE 2

Count the different

The table `orders` (`customer_id` is NULL for a guest checkout):

order_id	customer_id	product
1	7	pen
2	7	ink
3	9	pen
4	NULL	pen

Return one row with three counts: `order_count` (all orders), `buyer_count` (different known customers — guests are not counted) and `product_count` (different products).

Output: `order_count | buyer_count | product_count`.

```
order_count | buyer_count | product_count
4 | 2 | 2
```

INPUT 1

```
CREATE TABLE orders (
  order_id    INT PRIMARY KEY,
  customer_id INT,
  product     VARCHAR(40) NOT NULL
);
INSERT INTO orders VALUES
  (1, 7, 'pen'),
  (2, 7, 'ink'),
  (3, 9, 'pen'),
  (4, NULL, 'pen');
```

OUTPUT 1

```
order_count | buyer_count |
product_count
4 | 2 | 2
```

REFERENCE SOLUTION

```
SELECT COUNT(*)                   AS order_count,
       COUNT(DISTINCT customer_id) AS buyer_count,
       COUNT(DISTINCT product)     AS product_count
FROM orders;
```

QUIZ

1. What does `SELECT DISTINCT dept, job FROM employees` return?

A. Each department once

B. Each (dept, job) pair once

C. Each job once

D. An error: DISTINCT takes one column

2. `city` is 'Pune', NULL, NULL, 'Pune'. What do `SELECT DISTINCT city` and `SELECT COUNT(DISTINCT city)` give?

A. 2 rows; 2

B. 2 rows (NULL and Pune); 1

C. 3 rows; 3

D. 1 row; 1

3. What does `SELECT DISTINCT(city), name FROM customers` apply DISTINCT to?

A. city only

B. name only

C. the pair (city, name)

D. Nothing — it is a syntax error

4. Why does MySQL 8 refuse `SELECT DISTINCT city FROM customers ORDER BY joined`?

A. DISTINCT cannot be sorted

B. After DISTINCT merges rows, there is no single `joined` value to sort a city by

C. ORDER BY must come before DISTINCT

D. joined is a date

5. A join returns duplicate rows. What is the best first step?

A. Add DISTINCT

B. Find out why the join matches more rows than expected

C. Add LIMIT

D. Add GROUP BY on every column

ANSWER KEY · DISTINCT — REMOVING DUPLICATE ROWS

1. B DISTINCT compares whole result rows. A department appears again for each different job in it.

2. B DISTINCT collapses the NULLs into one NULL row, so two rows. COUNT(DISTINCT) ignores NULL entirely, so 1.

3. C The parentheses only surround the expression `city`; DISTINCT always applies to the whole row. Don't write it that way.

4. B Many customers share a city, each with their own join date. With DISTINCT, ORDER BY may only use selected columns.

5. B DISTINCT hides the symptom but the query still does the extra work, and aggregates over the same rows would be wrong. Usually the join condition is incomplete.

CHECKPOINT · 25 MIN

Checkpoint — Selecting and filtering

This checkpoint covers the whole module: column and table aliases and where they may be used; calculated columns with arithmetic, decimal division and `DIV`, `MOD`, `ROUND` and `CONCAT`; the `WHERE` clause with comparisons on numbers, text and dates; `AND`, `OR` and `NOT` with precedence and three-valued logic; `IN`, `BETWEEN` and `LIKE`; and `DISTINCT` with `COUNT(DISTINCT …)`.

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:

- Where may a column alias be used, and why not in `WHERE`?
- What do `7 / 2`, `7 DIV 2` and `7 % 2` return in MySQL, and what does `NULL + 1` return?
- Why does `WHERE dept <> 'Sales'` leave out the rows whose `dept` is NULL, and how do you include them?
- Which binds first, `AND` or `OR`, and where do the parentheses go?
- Is `BETWEEN` inclusive, and why is it risky on a `DATETIME` column?
- What do `%` and `_` match in `LIKE`, and how do you search for a literal `%`?
- What happens to `NOT IN` when its list contains NULL?
- How do `COUNT(*)`, `COUNT(col)` and `COUNT(DISTINCT col)` differ?

The three queries are a price list with renamed and calculated columns, a filtered order list that needs parentheses, a range and a NULL check, and a set of distinct values found with a `LIKE` pattern.

EXERCISE 1

A discounted price list

The table `products` (`discount_pct` is NULL when no discount is set):

product_id	name	price	discount_pct
1	Kettle	1999	10
2	Toaster	2500	0
3	Mixer	4499	15

Return `id` (`product_id`), `product` (`name`), `list_price` (`price`) and `sale_price`: the price less its discount percentage, rounded to 2 places. A NULL discount gives a NULL `sale_price`. Order by `sale_price` descending (NULLs last, which is MySQL's descending order), ties by `product_id`.

Output: `id | product | list_price | sale_price`.

```
id | product | list_price | sale_price
3 | Mixer | 4499 | 3824.15
2 | Toaster | 2500 | 2500
1 | Kettle | 1999 | 1799.1
```

INPUT 1

```
CREATE TABLE products (
  product_id   INT PRIMARY KEY,
  name         VARCHAR(60) NOT NULL,
  price        INT NOT NULL,
  discount_pct INT
);
INSERT INTO products VALUES
  (1, 'Kettle', 1999, 10),
  (2, 'Toaster', 2500, 0),
  (3, 'Mixer', 4499, 15);
```

OUTPUT 1

```
id | product | list_price | sale_price
3 | Mixer | 4499 | 3824.15
2 | Toaster | 2500 | 2500
1 | Kettle | 1999 | 1799.1
```

REFERENCE SOLUTION

```sql
SELECT product_id AS id,
       name       AS product,
       price      AS list_price,
       ROUND(price * (100 - discount_pct) / 100, 2) AS sale_price
FROM products
ORDER BY sale_price DESC, product_id;
```

EXERCISE 2

Orders to review

The table `orders`:

order_id	status	amount	country	coupon
1	pending	12000	IN	NULL
2	pending	3000	US	NULL
3	shipped	25000	IN	NULL
4	on_hold	15000	IN	SAVE10
5	on_hold	9999	IN	NULL

An order needs review when **all three** hold: its status is `pending` or `on_hold`; its amount is between 10000 and 50000 inclusive **or** it ships outside India (country is not `IN`); and no coupon was used (`coupon` is NULL). Return `order_id`, `status` and `amount` of those orders, ordered by `order_id`.

Output: `order_id | status | amount`.

```
order_id | status | amount
1 | pending | 12000
2 | pending | 3000
```

INPUT 1

```
CREATE TABLE orders (
  order_id INT PRIMARY KEY,
  status   VARCHAR(20) NOT NULL,
  amount   INT NOT NULL,
  country  VARCHAR(2) NOT NULL,
  coupon   VARCHAR(20)
);
INSERT INTO orders VALUES
  (1, 'pending', 12000, 'IN', NULL),
  (2, 'pending', 3000, 'US', NULL),
  (3, 'shipped', 25000, 'IN', NULL),
  (4, 'on_hold', 15000, 'IN', 'SAVE10'),
  (5, 'on_hold', 9999, 'IN', NULL);
```

OUTPUT 1

```
order_id | status | amount
1 | pending | 12000
2 | pending | 3000
```

REFERENCE SOLUTION

```sql
SELECT order_id, status, amount
FROM orders
WHERE status IN ('pending', 'on_hold')
  AND (amount BETWEEN 10000 AND 50000 OR country <> 'IN')
  AND coupon IS NULL
ORDER BY order_id;
```

EXERCISE 3

Departments with engineers

The table `employees` (`dept` may be NULL):

emp_id	name	job_title	dept
1	Asha	Software Engineer	Platform
2	Ravi	Senior Engineer	Platform
3	Meera	Data Engineer	Analytics
4	Kabir	Analyst	Analytics
5	Dev	QA Engineer	NULL

Return each **department** that has at least one employee whose `job_title` contains the text `Engineer`, once each, in alphabetical order. Leave out a NULL department.

Output: one column `dept`.

```
dept
Analytics
Platform
```

INPUT 1

```
CREATE TABLE employees (
  emp_id    INT PRIMARY KEY,
  name      VARCHAR(40) NOT NULL,
  job_title VARCHAR(60) NOT NULL,
  dept      VARCHAR(20)
);
INSERT INTO employees VALUES
  (1, 'Asha', 'Software Engineer',
'Platform'),
  (2, 'Ravi', 'Senior Engineer',
'Platform'),
  (3, 'Meera', 'Data Engineer',
'Analytics'),
  (4, 'Kabir', 'Analyst', 'Analytics'),
  (5, 'Dev', 'QA Engineer', NULL);
```

OUTPUT 1

```
dept
Analytics
Platform
```

REFERENCE SOLUTION

```sql
SELECT DISTINCT dept
FROM employees
WHERE job_title LIKE '%Engineer%'
  AND dept IS NOT NULL
ORDER BY dept;
```

CHECKPOINT QUESTIONS

1. Which query is refused by MySQL?

A. `SELECT price * 2 AS p FROM t ORDER BY p`

B. `SELECT price * 2 AS p FROM t WHERE p > 10`

C. `SELECT price * 2 AS p FROM t WHERE price * 2 > 10`

D. `SELECT price AS p FROM t ORDER BY price`

2. In MySQL, what is `SELECT 10 / 4 AS a, 10 DIV 4 AS b, 10 % 4 AS c;` ?

A. 2.5000, 2, 2

B. 2, 2, 2

C. 2.5000, 2.5000, 2

D. 2, 2.5000, 2

3. What is `CONCAT('Order-', NULL)` in MySQL?

A. `Order-`

B. `Order-NULL`

C. NULL

D. An error

4. Which condition returns the rows whose `manager_id` is missing?

A. `manager_id = NULL`

B. `manager_id IS NULL`

C. `manager_id = ''`

D. `NOT manager_id`

5. What does `WHERE category = 'books' OR category = 'games' AND price < 500` return?

A. Books and games under 500

B. All books, plus games under 500

C. Only games under 500

D. Nothing

6. What is `TRUE OR NULL`?

A. TRUE

B. NULL

C. FALSE

D. An error

7. Which is the same as `x NOT BETWEEN 10 AND 20`?

A. `x < 10 AND x > 20`

B. `x < 10 OR x > 20`

C. `x ⩽ 10 OR x ⩾ 20`

D. `x > 10 AND x < 20`

8. Which names does `LIKE 'S____'` (S and four underscores) match?

A. Every name starting with S

B. Names of exactly five characters starting with S

C. Names of at least five characters starting with S

D. Names ending in S

9. Why might `WHERE id NOT IN (SELECT manager_id FROM employees)` suddenly return no rows?

A. The subquery returned duplicates

B. One employee has a NULL manager_id

C. The subquery is too large

D. NOT IN needs DISTINCT

10. Rows have `product` = pen, pen, ink, NULL. What is `COUNT(product)`?

A. 4

B. 3

C. 2

D. 1

11. Which query returns each department once?

A. `SELECT DISTINCT dept, name FROM employees`

B. `SELECT DISTINCT dept FROM employees`

C. `SELECT dept FROM employees LIMIT 1`

D. `SELECT COUNT(DISTINCT dept) FROM employees`

12. What does MySQL do with `SELECT 100 / 0;`?

A. Returns 0

B. Returns NULL (with a warning)

C. Returns infinity

D. Always raises an error

ANSWER KEY · CHECKPOINT — SELECTING AND FILTERING

1. B WHERE is evaluated before SELECT, so the alias `p` is unknown there. ORDER BY can use it; repeating the expression in WHERE works.

2. A `/` is decimal division in MySQL, `DIV` integer division, `%` the remainder.

3. C CONCAT returns NULL if any argument is NULL. `CONCAT_WS` skips NULL arguments instead.

4. B Only `IS NULL` tests for a missing value. `= NULL` is never true; an empty string is a value; `NOT manager_id` is unknown for NULL.

5. B AND binds first: `books OR (games AND price < 500)`. Parenthesise the OR group to get the intended filter.

6. A TRUE OR anything is TRUE, so the unknown does not matter. `FALSE OR NULL` would be NULL.

7. B BETWEEN is `x ⩾ 10 AND x ⩽ 20`; by De Morgan its negation is `x < 10 OR x > 20`. The AND version can never be true.

8. B Each `_` is exactly one character, so the pattern matches S plus four more: 'Sunil', 'Sagar' — not 'Sameer' or 'Sam'.

9. B A NULL in the list makes `id <> NULL` unknown for every row, so NOT IN is never true. Filter NULLs out of the subquery or use NOT EXISTS.

10. B `COUNT(col)` counts non-NULL values, duplicates included: pen, pen, ink. `COUNT(*)` would be 4 and `COUNT(DISTINCT product)` 2.

11. B DISTINCT on just `dept`. Adding `name` makes each (dept, name) pair distinct; COUNT returns a number, not the departments.

12. B MySQL returns NULL for division by zero (an error only under strict settings on writes). PostgreSQL and SQL Server raise an error.

CodeKairo · SQL study plan · Module 2: Selecting and filtering. Solve the exercises online to track your progress and earn XP.