Module 4 of 16

NULLs and expressions

NULL as a missing value, UNKNOWN and three-valued logic, IS NULL and MySQL's NULL-safe equality, COALESCE, IFNULL and NULLIF, searched and simple CASE, data types, DIV versus decimal division, implicit conversion and CAST.

Real tables have gaps, and SQL marks every gap with one thing: `NULL`. It is not zero and not an empty string, and it follows rules of its own — arithmetic with it gives NULL, comparisons with it give a third truth value, UNKNOWN, and `WHERE` keeps only rows that are TRUE. Those rules explain more wrong query results than anything else in SQL, and they are a fixture of interviews: `NULL = NULL`, `NOT IN` with a NULL, `COUNT(*)` versus `COUNT(col)`. This module teaches the rules, the tools that test for and replace NULLs, the `CASE` expression that lets a query make decisions, and the types and conversions every expression runs on. The examples are MySQL 8, with the spellings of SQL Server, Oracle and PostgreSQL called out where interviewers ask about them.

NULL and three-valued logic covers what NULL means, how it spreads through expressions, UNKNOWN, the truth tables for AND, OR and NOT, why rows vanish from both `=` and `<>`, and the `NOT IN` trap. IS NULL covers `IS NULL` / `IS NOT NULL`, the `= NULL` mistake, parentheses with OR, MySQL's `<=>`, `IS DISTINCT FROM` and the `IS TRUE` family. COALESCE, IFNULL and NULLIF covers defaults, the other databases' `ISNULL` and `NVL`, turning placeholders into NULL and guarding a division. CASE covers searched and simple forms, first-match-wins, ELSE and NULL, CASE in WHERE and ORDER BY, and MySQL's `IF()`. Types and CAST covers the MySQL types, DECIMAL versus FLOAT, `/` versus `DIV`, implicit conversion and `CAST` / `CONVERT`.

The exercises are queries against small tables full of gaps: line totals with missing quantities, everyone outside one department, incomplete contact details, prices that changed, display names with fallbacks, a click-through rate that never divides by zero, grade bands, an order-status board in a custom order, ticket codes sorted as numbers and boxes packed with DIV and MOD. The checkpoint adds a delivery fee with an optional discount, a contact list that treats blanks as missing, and an audit that labels every change.

CONTENTS

1	NULL and three-valued logic	15 min
2	IS NULL, IS NOT NULL and NULL-safe equality	13 min

3	COALESCE, IFNULL and NULLIF	14 min
4	CASE expressions — if-then-else in SQL	15 min
5	Data types, implicit conversion and CAST	15 min
✓	Checkpoint — NULLs and expressions	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 · 15 MIN

NULL and three-valued logic

Real data has gaps. A customer never gave a phone number, a product has not been priced yet, an employee has no manager because she runs the company, a sensor missed a reading. SQL represents every one of those gaps with a single marker, `NULL`, and gives it rules unlike any other value. Most surprising query results — rows that vanish from a filter, totals that come out empty, a `NOT IN` that returns nothing — trace back to those rules. This lesson explains what NULL is, how it spreads through expressions, and **three-valued logic**, the reason conditions involving NULL are neither true nor false.

NULL is not a value

`NULL` is a marker that says "there is no value here". It is not zero, not the empty string `''`, not `false` and not the text `'NULL'`:

Stored	Meaning
`0`	The discount is known: zero.
`''`	The middle name is known: there isn't one.
`NULL`	Nobody recorded the discount (or the middle name).

That distinction matters. "Average discount" should count a known zero and ignore an unknown; "customers without a middle name" is a different list from "customers whose middle name we never asked for". A column declared `NOT NULL` can never hold the marker, which is why schema design (a later module) cares so much about which columns allow it. One dialect note interviewers like: Oracle stores an empty string as NULL, so in Oracle `''` *is* NULL. MySQL, PostgreSQL, SQL Server and SQLite keep the two apart.

NULL spreads through expressions

If a value is unknown, anything computed from it is unknown too:

```sql
SELECT 5 + NULL      AS a,   -- NULL
       NULL * 0      AS b,   -- NULL, even times zero
       10 / NULL     AS c,   -- NULL
       CONCAT('Hi ', NULL) AS d;  -- NULL in MySQL
```

a	b	c	d
NULL	NULL	NULL	NULL

So a line total `qty * unit_price` is NULL whenever either factor is missing, and a full name built with `CONCAT(first_name, ' ', last_name)` is NULL for everyone without a last name. (PostgreSQL's `||` behaves the same; `CONCAT` in PostgreSQL and SQL Server skips NULLs instead — one of the dialect differences that bites when queries move between databases.) The functions in the next lessons exist largely to stop this spread on purpose.

Comparisons give UNKNOWN

What is `NULL = 5`? The left side is unknown, so whether it equals 5 is unknown. SQL's answer is neither TRUE nor FALSE but a third truth value, **UNKNOWN**. MySQL displays UNKNOWN as `NULL`, because its booleans are numbers (`1` true, `0` false) and the unknown one is NULL:

```sql
SELECT 5 = 5     AS t,     -- 1
       5 = 6     AS f,     -- 0
       NULL = 5  AS u1,    -- NULL (unknown)
       NULL = NULL AS u2,  -- NULL — not 1!
       NULL <> 5 AS u3;    -- NULL
```

`NULL = NULL` being unknown is the rule people find strangest, and it follows directly: two unknown values cannot be known to be equal. Every comparison operator — `=`, `<>`, `≠`, `<`, `>`, `⩽`, `⩾`, `BETWEEN`, `LIKE`, `IN` — returns UNKNOWN when a NULL is involved in the comparison it makes.

WHERE keeps only TRUE

A `WHERE` clause keeps a row when its condition is **TRUE**. FALSE and UNKNOWN are both thrown away. That one sentence explains most NULL surprises:

```sql
SELECT name FROM employees WHERE dept = 'Sales';     -- NULL-dept rows: unknown
SELECT name FROM employees WHERE dept <> 'Sales';    -- NULL-dept rows: unknown
```

Run both and add up the rows: employees with no department appear in neither. They are not "Sales" and not "not Sales", as far as SQL can tell. If the business meaning is "everyone except Sales, including people with no department", write the NULL case out:

```sql
SELECT name
FROM employees
WHERE dept <> 'Sales' OR dept IS NULL;
```

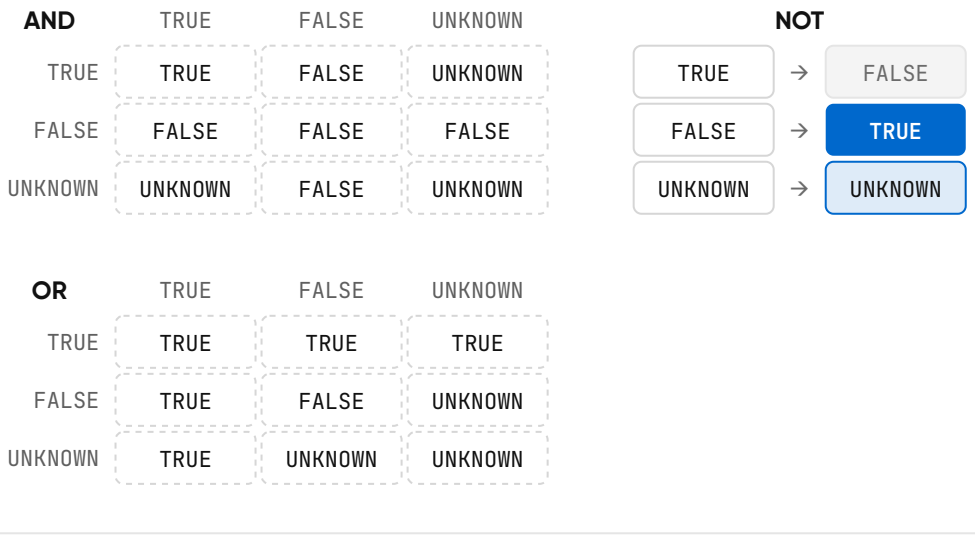
`IS NULL` is the one test that answers a plain yes or no about NULL — it gets its own lesson next. For now remember that `dept = NULL` is not a test at all: it is UNKNOWN for every row, so `WHERE dept = NULL` returns nothing, silently.

The three-valued truth tables

When conditions are combined, UNKNOWN combines by a simple principle: **if the known parts already decide the answer, the unknown part does not matter; otherwise the answer is unknown.**

`AND` is false as soon as one side is false, and `OR` is true as soon as one side is true:

Three-valued logic: AND, OR and NOT with UNKNOWN



1. AND is TRUE only when both sides are TRUE. With an UNKNOWN on either side the answer is usually UNKNOWN too — but not always.
2. OR is TRUE when either side is TRUE. In both tables each row is the left value and each column the right one.
3. The principle: if the known side already decides the answer, the unknown side does not matter. FALSE AND anything is FALSE; TRUE OR anything is TRUE.
4. Everywhere else UNKNOWN wins, and WHERE drops UNKNOWN just as it drops FALSE. That is how a NULL in a condition makes rows disappear.
5. NOT swaps TRUE and FALSE but leaves UNKNOWN unknown — so wrapping a condition in NOT never brings a NULL row back.

And `NOT UNKNOWN` is UNKNOWN — negating an unknown fact does not make it known. That is why wrapping a condition in `NOT` never brings NULL rows back:

```sql
-- Neither of these returns rows whose discount is NULL
SELECT name FROM products WHERE discount > 10;
SELECT name FROM products WHERE NOT (discount > 10);
```

In MySQL you can see the tables directly: `SELECT NULL AND 0, NULL OR 1, NULL AND 1, NOT NULL` returns `0, 1, NULL, NULL`.

The NOT IN trap

`IN` is a chain of `=` joined with `OR`, so `x IN (1, 2, NULL)` means `x = 1 OR x = 2 OR x = NULL`. For `x = 1` that is TRUE (one TRUE in an OR decides it). For `x = 3` it is `FALSE OR FALSE OR UNKNOWN` = UNKNOWN.

`NOT IN` flips it into a chain of `<>` joined with `AND`: `x <> 1 AND x <> 2 AND x <> NULL`. The last part is always UNKNOWN, so the whole condition can never be TRUE:

```sql
SELECT id FROM orders WHERE customer_id NOT IN (1, 2, NULL);   -- no rows, eve
```

With literal lists you would rarely write that NULL, but it arrives the moment the list comes from a subquery over a nullable column — the famous "NOT IN returns nothing" bug, which the subqueries module covers properly.

Where else NULL behaves differently

Some parts of SQL treat NULLs as *alike* rather than unknown — a practical exception to the comparison rule:

- `DISTINCT` keeps one NULL, treating all NULLs as duplicates of each other.
- `ORDER BY` puts all NULLs together (first ascending in MySQL, as you saw).
- `GROUP BY` (later) puts all NULLs in one group.
- Aggregates such as `SUM` and `AVG` skip NULLs; `COUNT(col)` counts only non-NULL values while `COUNT(*)` counts rows.

Knowing which rule applies where — "unknown" in comparisons, "the same" in grouping and sorting — is the whole skill.

Before you move on

- NULL means "no value"; it is not 0, `''` or false.
- Arithmetic and most functions with a NULL input give NULL.
- Every comparison with NULL, including `NULL = NULL`, gives UNKNOWN.
- WHERE keeps only TRUE rows, so NULL rows fall out of both `=` and `<>`.
- AND with FALSE is FALSE, OR with TRUE is TRUE, everything else touching UNKNOWN stays UNKNOWN.

EXERCISE 1

Line totals with missing data

The `order_lines` table; some quantities and prices were never recorded:

id	item	qty	unit_price
1	pen	10	5
2	lamp	NULL	450
3	desk	1	NULL
4	mug	30	4

For every line return `id`, `item`, `total` (= `qty * unit_price`) and `is_large` (the comparison `qty * unit_price > 100`, which MySQL shows as 1 or 0), ordered by `id`. Do not replace the missing values — let NULL show where the answer is unknown.

```
id | item | total | is_large
1 | pen | 50 | 0
2 | lamp | NULL | NULL
3 | desk | NULL | NULL
4 | mug | 120 | 1
```

INPUT 1

```
CREATE TABLE order_lines (id INT PRIMARY
KEY, item VARCHAR(40), qty INT,
unit_price INT);
INSERT INTO order_lines VALUES (1,
'pen', 10, 5), (2, 'lamp', NULL, 450),
(3, 'desk', 1, NULL), (4, 'mug', 30, 4);
```

OUTPUT 1

```
id | item | total | is_large
1 | pen | 50 | 0
2 | lamp | NULL | NULL
3 | desk | NULL | NULL
4 | mug | 120 | 1
```

REFERENCE SOLUTION

```sql
SELECT id,
       item,
       qty * unit_price AS total,
       qty * unit_price > 100 AS is_large
FROM order_lines
ORDER BY id;
```

EXERCISE 2

Everyone outside Sales

The `employees` table; `dept` is NULL for people not yet assigned:

id	name	dept
1	Asha	Sales
2	Ben	NULL
3	Chen	Support
4	Dina	Sales
5	Eli	Ops

HR wants everyone who is **not in Sales — including people with no department**. Return `id`, `name`, `dept` ordered by `id`.

```
id | name | dept
2 | Ben | NULL
3 | Chen | Support
5 | Eli | Ops
```

INPUT 1

```
CREATE TABLE employees (id INT PRIMARY
KEY, name VARCHAR(40), dept
VARCHAR(20));
INSERT INTO employees VALUES (1, 'Asha',
'Sales'), (2, 'Ben', NULL), (3, 'Chen',
'Support'), (4, 'Dina', 'Sales'), (5,
'Eli', 'Ops');
```

OUTPUT 1

```
id | name | dept
2 | Ben | NULL
3 | Chen | Support
5 | Eli | Ops
```

REFERENCE SOLUTION

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

QUIZ

1. What does `SELECT NULL = NULL` return in MySQL?

A. 1

B. 0

C. NULL

D. An error

2. A table has 10 employees: 4 in Sales, 4 in other departments, 2 with a NULL dept. How many rows does `WHERE dept <> 'Sales'` return?

A. 6

B. 4

C. 10

D. 2

3. What is `NULL AND FALSE` ?

A. TRUE

B. FALSE

C. UNKNOWN

D. An error

4. What does `SELECT id FROM t WHERE x NOT IN (1, 2, NULL)` return?

A. Rows where x is 3 or more

B. Rows where x is NULL

C. No rows, ever

D. Every row

5. In MySQL, what is `CONCAT('Order #', NULL)`?

A. 'Order #'

B. 'Order #NULL'

C. NULL

D. An error

ANSWER KEY · NULL AND THREE-VALUED LOGIC

1. C Two unknown values cannot be known to be equal, so the comparison is UNKNOWN, which MySQL displays as NULL. It is not 1 (true) and not 0 (false). Use `IS NULL` to test for NULL.

2. B For the 2 NULL rows the comparison is UNKNOWN, and WHERE keeps only TRUE, so only the 4 other-department rows remain. The NULL rows appear in neither `= 'Sales'` nor `<> 'Sales'`.

3. B AND is false as soon as one side is false, whatever the other side is. `NULL AND TRUE`, by contrast, is UNKNOWN, because the unknown side could decide it.

4. C `NOT IN` is `x <> 1 AND x <> 2 AND x <> NULL`, and the last part is always UNKNOWN, so the AND can be FALSE or UNKNOWN but never TRUE.

5. C MySQL's CONCAT returns NULL if any argument is NULL — the NULL spreads. PostgreSQL's and SQL Server's CONCAT skip NULLs instead, which would give 'Order #'.

LESSON 2 · 13 MIN

IS NULL, IS NOT NULL and NULL-safe equality

The previous lesson showed that `= NULL` is never true. So how *do* you find the rows where a value is missing? SQL has a dedicated predicate for exactly that, `IS NULL`, and its opposite, `IS NOT NULL`. They are the most used NULL tools in everyday SQL, and the quickest way to fail an interview question is to write `= NULL` instead. This lesson covers both predicates, combining them with other conditions, MySQL's NULL-safe equality operator `<=>`, the standard's `IS DISTINCT FROM`, the `IS TRUE` family, and the classic "which values changed" query.

IS NULL and IS NOT NULL

```sql
-- Customers who never gave a phone number
SELECT id, name
FROM customers
WHERE phone IS NULL;

-- Customers we can call
SELECT id, name, phone
FROM customers
WHERE phone IS NOT NULL;
```

Unlike a comparison, `IS NULL` never returns UNKNOWN: it is TRUE when the expression has no value and FALSE when it has one. Together, `IS NULL` and `IS NOT NULL` split a table cleanly into two parts with no row left over — which `=` and `<>` cannot do.

Both predicates work on any expression, not just a column. Because NULL spreads through arithmetic, `qty * unit_price IS NULL` is true when either factor is missing, and `CONCAT(first_name, last_name) IS NULL` is true when either name is (in MySQL).

`IS NOT NULL` is also how you *exclude* missing values before ranking or comparing:

```sql
-- The three smallest real discounts (NULLs would otherwise sort first)
SELECT name, discount
FROM products
WHERE discount IS NOT NULL
ORDER BY discount, id
LIMIT 3;
```

The `= NULL` mistake

```sql
SELECT id FROM customers WHERE phone = NULL;      -- always empty
SELECT id FROM customers WHERE phone ≠ NULL;      -- also always empty
```

Neither statement is an error. MySQL runs them happily and returns no rows, because the condition is UNKNOWN for every row. That silence is what makes the bug dangerous: a report built on `= NULL` looks fine and is simply missing data. The same mistake hides in an `UPDATE … WHERE col = NULL` that updates nothing. Make `IS NULL` a reflex.

(SQL Server has a legacy setting, `ANSI_NULLS OFF`, under which `= NULL` worked; it is deprecated, and nothing in MySQL behaves that way.)

Combining with other conditions

`IS NULL` usually appears inside a larger condition, and the three-valued rules decide how it combines. The pattern you met in the previous lesson keeps NULL rows on purpose:

```sql
-- Everyone not in Sales, including people with no department
WHERE dept <> 'Sales' OR dept IS NULL
```

For a NULL dept the first part is UNKNOWN and the second TRUE, and `UNKNOWN OR TRUE` is TRUE. Mind the parentheses when you mix `AND` and `OR`, because `AND` binds tighter:

```sql
-- Wrong: means  city = 'Pune' AND phone IS NULL,  OR  email IS NULL (any city
WHERE city = 'Pune' AND phone IS NULL OR email IS NULL

-- Right: Pune customers missing a phone or an email
WHERE city = 'Pune' AND (phone IS NULL OR email IS NULL)
```

`IS NULL` also makes a good sort key. In MySQL `col IS NULL` is `1` or `0`, which is how `ORDER BY col IS NULL, col` pushed NULLs to the end in the sorting module; `ORDER BY (email IS NULL AND phone IS NULL) DESC` puts customers missing both at the top.

NULL-safe equality: `<=>`

Sometimes you want two NULLs to count as equal — when comparing an old and a new version of a row, for instance. MySQL has an operator for it, `⟺`, sometimes called the spaceship operator:

a	b	`a = b`	`a ⟺ b`
5	5	1	1
5	6	0	0
5	NULL	NULL	0
NULL	NULL	NULL	1

`⟺` never returns NULL. That makes "did this value change?" a one-liner:

```
-- Products whose price changed, where NULL → 80 and 70 → NULL both count as c
SELECT sku, old_price, new_price
FROM price_changes
WHERE NOT (old_price ⟺ new_price);
```

With plain `<>`, the rows that went from NULL to a price, or from a price to NULL, would vanish (`NULL <> 80` is UNKNOWN). Without `⟺` you would need the long form:

```
WHERE old_price <> new_price
   OR (old_price IS NULL AND new_price IS NOT NULL)
   OR (old_price IS NOT NULL AND new_price IS NULL)
```

`col ⟺ NULL` is also a valid, if unusual, way of writing `col IS NULL`.

The standard spelling: IS DISTINCT FROM

The SQL standard expresses the same idea in words: `a IS DISTINCT FROM b` (NULL-safe "not equal") and `a IS NOT DISTINCT FROM b` (NULL-safe "equal"). PostgreSQL, SQLite and SQL Server 2022 support them; MySQL does not, and uses `<=>`. In an interview, naming both shows you know the problem rather than one vendor's syntax.

MySQL's ISNULL() and the IS TRUE family

MySQL has a function form, `ISNULL(expr)`, returning `1` when the expression is NULL and `0` otherwise — the same as `expr IS NULL`. Do not confuse it with SQL Server's `ISNULL(a, b)`, which takes two arguments and means "a, or b if a is NULL" (MySQL calls that `IFNULL`, next lesson). Same name, different function — a favourite trick question.

The standard also lets you test a condition's truth value directly: `cond IS TRUE`, `IS FALSE`, `IS UNKNOWN`, and their `IS NOT` forms. They never return UNKNOWN themselves:

```
SELECT (NULL > 1) IS UNKNOWN,    -- 1
       (NULL > 1) IS NOT TRUE,   -- 1
       (5 > 1) IS TRUE;          -- 1
```

`WHERE (discount > 10) IS NOT TRUE` keeps rows where the discount is 10 or less *and* rows where it is NULL — the honest complement of `WHERE discount > 10`. MySQL supports all of these; they are rare in practice but show up in theory questions.

Before you move on

- Test for missing values with `IS NULL` / `IS NOT NULL`, never `= NULL`.
- `IS NULL` is never UNKNOWN, so it splits rows cleanly.
- Parenthesise `OR` when you mix it with `AND`.
- MySQL's `<=>` is NULL-safe equality; `NOT (a <=> b)` finds changed values.
- The standard spells it `IS [NOT] DISTINCT FROM`; MySQL's `ISNULL(x)` is not SQL Server's `ISNULL(a, b)`.

EXERCISE 1

Incomplete contact details

The `customers` table:

id	name	email	phone
1	Asha	asha@mail.in	98100
2	Ben	NULL	98200
3	Chen	NULL	NULL
4	Dina	dina@mail.in	NULL
5	Eli	eli@mail.in	98500
6	Fay	NULL	NULL

List every customer missing an email, a phone or both, as `id`, `name`, `email`, `phone`. Customers missing **both** come first; otherwise order by `id`. An empty string is a value, not a missing one.

```
id | name | email | phone
3 | Chen | NULL | NULL
6 | Fay | NULL | NULL
2 | Ben | NULL | 98200
4 | Dina | dina@mail.in | NULL
```

INPUT 1

```
CREATE TABLE customers (id INT PRIMARY
KEY, name VARCHAR(40), email
VARCHAR(60), phone VARCHAR(20));
INSERT INTO customers VALUES (1, 'Asha',
'asha@mail.in', '98100'), (2, 'Ben',
NULL, '98200'), (3, 'Chen', NULL, NULL),
(4, 'Dina', 'dina@mail.in', NULL), (5,
'Eli', 'eli@mail.in', '98500'), (6,
'Fay', NULL, NULL);
```

OUTPUT 1

```
id | name | email | phone
3 | Chen | NULL | NULL
6 | Fay | NULL | NULL
2 | Ben | NULL | 98200
4 | Dina | dina@mail.in | NULL
```

REFERENCE SOLUTION

```sql
SELECT id, name, email, phone
FROM customers
WHERE email IS NULL OR phone IS NULL
ORDER BY (email IS NULL AND phone IS NULL) DESC, id;
```

EXERCISE 2

Prices that changed

The `price_changes` table compares yesterday's and today's price of each product; NULL means "no price":

sku	old_price	new_price
A1	100	120
A2	50	50
A3	NULL	80
A4	70	NULL
A5	NULL	NULL
A6	0	0

Return the products whose price **changed** as `sku`, `old_price`, `new_price`, ordered by `sku`. Gaining or losing a price counts as a change; NULL to NULL does not.

```
sku | old_price | new_price
A1 | 100 | 120
A3 | NULL | 80
A4 | 70 | NULL
```

INPUT 1

```
CREATE TABLE price_changes (sku
VARCHAR(10) PRIMARY KEY, old_price INT,
new_price INT);
INSERT INTO price_changes VALUES ('A1',
100, 120), ('A2', 50, 50), ('A3', NULL,
80), ('A4', 70, NULL), ('A5', NULL,
NULL), ('A6', 0, 0);
```

OUTPUT 1

```
sku | old_price | new_price
A1 | 100 | 120
A3 | NULL | 80
A4 | 70 | NULL
```

REFERENCE SOLUTION

```
SELECT sku, old_price, new_price
FROM price_changes
WHERE NOT (old_price ⟺ new_price)
ORDER BY sku;
```

QUIZ

1. What does `SELECT * FROM customers WHERE phone = NULL` return?

A. Customers without a phone

B. Every customer

C. No rows, without an error

D. A syntax error

2. What is `5 <=> NULL` in MySQL?

A. NULL

B. 0

C. 1

D. An error

3. Which WHERE finds Pune customers missing a phone or an email?

A. `WHERE city = 'Pune' AND phone IS NULL OR email IS NULL`

B. `WHERE city = 'Pune' AND (phone IS NULL OR email IS NULL)`

C. `WHERE city = 'Pune' AND (phone OR email) IS NULL`

D. `WHERE city = 'Pune' AND phone, email IS NULL`

4. In MySQL, what does `ISNULL(discount)` return for a row whose discount is NULL?

A. NULL

B. 1

C. 0

D. The second argument

5. A `middle_name` column holds `''` for people with no middle name. Which rows does `WHERE middle_name IS NULL` return?

A. The rows with `''`

B. No rows, unless some middle names are actually NULL

C. Every row

D. An error, because `''` is not NULL

ANSWER KEY · IS NULL, IS NOT NULL AND NULL-SAFE EQUALITY

1. C `phone = NULL` is UNKNOWN for every row, so WHERE keeps none — and MySQL reports no error, which is why the bug survives. `phone IS NULL` is the test.

2. B The NULL-safe operator never returns NULL: it is 1 when both sides are NULL or equal, and 0 otherwise. `5 = NULL` would be NULL.

3. B AND binds tighter than OR, so the first option also returns customers from every city whose email is NULL. The parentheses keep the city filter on both cases. The last two are not valid ways to test two columns.

4. B MySQL's ISNULL takes one argument and returns 1 or 0, like `discount IS NULL`. SQL Server's ISNULL(a, b) is a different function — a default value, which MySQL calls IFNULL.

5. B An empty string is a value, so IS NULL is false for it. To treat blanks as missing, test `middle_name IS NULL OR middle_name = ''` or use `NULLIF(middle_name, '')`. (Oracle, which stores '' as NULL, is the exception.)

LESSON 3 · 14 MIN

COALESCE, IFNULL and NULLIF

Knowing that NULLs spread through expressions and fall out of filters is half the story. The other half is controlling them: showing a sensible default instead of a blank, treating a missing quantity as zero in a sum, or — the opposite direction — turning a meaningless placeholder like an empty string or a zero divisor *into* NULL so it stops causing damage. Three small functions do all of this: `COALESCE`, `IFNULL` and `NULLIF`. They appear in nearly every real reporting query and in many interview answers, and their cousins in other databases (`ISNULL`, `NVL`) are a reliable source of trick questions.

COALESCE: the first non-NULL value

`COALESCE(a, b, c, …)` looks at its arguments from left to right and returns the first one that is not NULL. If every argument is NULL, the result is NULL.

```
SELECT id,
       COALESCE(nickname, first_name, 'Guest') AS display_name
FROM users;
```

id	display_name
1	ace
2	Bela
3	Guest

User 1 has a nickname, user 2 has only a first name, user 3 has neither. COALESCE reads like "use this, or else that, or else this default" — a chain of fallbacks of any length.

The commonest use is a default for display or arithmetic:

```
-- Treat a missing bonus as no bonus
SELECT name, salary + COALESCE(bonus, 0) AS total_pay
FROM employees;
```

Without COALESCE, `salary + bonus` would be NULL for everyone whose bonus was never set. Two cautions. First, the substitution exists only in the query's output; the table still holds NULL. Second, choose the default deliberately: replacing a missing test score with 0 makes the student look like they failed, and averaging those zeros drags the class average down. Sometimes NULL *is* the honest answer.

COALESCE is evaluated lazily — it stops at the first non-NULL argument — and it is standard SQL, supported by MySQL, PostgreSQL, SQL Server, Oracle and SQLite alike. When in doubt, use it.

IFNULL: the two-argument shortcut

`IFNULL(a, b)` is MySQL's name for the two-argument case: `a` unless `a` is NULL, in which case `b`.

```sql
SELECT name, IFNULL(phone, 'no phone') AS phone
FROM customers;
```

It is exactly `COALESCE(a, b)`. Every database has its own spelling for this, and interviewers love to ask which is which:

Database	Two-argument form	Standard form
MySQL, SQLite	`IFNULL(a, b)`	`COALESCE(…)`
SQL Server	`ISNULL(a, b)`	`COALESCE(…)`
Oracle	`NVL(a, b)`	`COALESCE(…)`
PostgreSQL	—	`COALESCE(…)`

Note the clash: in MySQL `ISNULL(x)` takes **one** argument and returns 1 or 0, while in SQL Server `ISNULL(a, b)` takes two and returns a value. Writing `COALESCE` avoids the whole question.

NULLIF: turning a value into NULL

`NULLIF(a, b)` goes the other way. It returns NULL when `a = b`, and `a` otherwise:

```sql
SELECT NULLIF(5, 5),    -- NULL
       NULLIF(5, 3),    -- 5
       NULLIF('', ''),  -- NULL
       NULLIF(NULL, 1); -- NULL (a was already NULL)
```

Why would you *want* a NULL? Because many systems store placeholders that mean "missing" — an empty string from a web form, `0` or `-1` for "not measured", `'N/A'` typed by hand. NULLIF turns them into a real NULL, so the NULL rules (and the COALESCE defaults) apply to them too:

```sql
-- Blank nicknames should fall back to the first name as well
SELECT COALESCE(NULLIF(nickname, ''), first_name, 'Guest') AS display_name
FROM users;
```

Read it inside out: an empty nickname becomes NULL, and then COALESCE skips it. The `COALESCE(NULLIF(x, ''), default)` combination is one of the most useful idioms in SQL.

Guarding a division with NULLIF

The other classic use of NULLIF is protecting a division:

```sql
SELECT name,
       ROUND(clicks / NULLIF(views, 0), 2) AS ctr
FROM campaigns;
```

When `views` is 0, `NULLIF(views, 0)` is NULL, and `clicks / NULL` is NULL — a campaign with no views has no click-through rate, rather than an infinite one. How much this matters depends on the database:

- **MySQL** returns NULL for `x / 0` in a SELECT (with a warning), so the guard is not strictly needed there — but data-changing statements in strict mode can raise an error instead.
- **PostgreSQL, SQL Server and Oracle** raise a *division by zero* error that aborts the whole query.

So `/ NULLIF(d, 0)` is the portable habit, and writing it shows an interviewer you have been bitten before. If the report should show 0 instead of NULL, wrap the whole thing: `COALESCE(clicks / NULLIF(views, 0), 0)`. And if the numerator might be NULL as well — clicks never recorded — decide what that means: `COALESCE(clicks, 0) / NULLIF(views, 0)` treats it as zero clicks.

Mixing types

The result of COALESCE takes a type that can hold all of its arguments. Mixing a number with a string makes the result a string in MySQL, which then sorts as text:

```
SELECT COALESCE(score, 'absent') AS score FROM results;  -- a string column na
```

That is fine for display and bad for sorting or arithmetic. Keep the arguments of one type where you can, and use a separate column or a `CASE` expression (next lesson) when a numeric column needs a text label.

Outside the SELECT list

Like every function, these work anywhere an expression is allowed. In `ORDER BY`, COALESCE decides where NULLs sort by giving them a stand-in value:

```
-- Products without a discount sort as if their discount were 0
SELECT name, discount
FROM products
ORDER BY COALESCE(discount, 0) DESC, id;
```

In `WHERE`, it folds the NULL case into a comparison. `WHERE COALESCE(dept, '') <> 'Sales'` keeps employees with no department, because their NULL becomes `''`, which is not `'Sales'` — a shorter spelling of `dept <> 'Sales' OR dept IS NULL`. Shorter is not always better here: wrapping a column in a function stops MySQL from using an index on that column for the comparison, so on a large table the explicit `OR … IS NULL` form is usually the faster one. The indexes module comes back to this rule, which has a name: keep predicates *sargable*.

COALESCE is a CASE in disguise

The standard defines COALESCE as shorthand for a CASE expression, which the next lesson teaches:

```sql
COALESCE(a, b, c)
-- means
CASE WHEN a IS NOT NULL THEN a
     WHEN b IS NOT NULL THEN b
     ELSE c
END
```

and `NULLIF(a, b)` as `CASE WHEN a = b THEN NULL ELSE a END`. Knowing the expansion helps when a rule is more complicated than "first non-NULL" — a CASE can test any condition, not only NULL — and it answers the interview question "how would you write COALESCE without COALESCE?".

Before you move on

- `COALESCE(a, b, …)` returns the first non-NULL argument; it is standard everywhere.
- `IFNULL(a, b)` is MySQL's two-argument form; SQL Server says `ISNULL`, Oracle `NVL`.
- `NULLIF(a, b)` returns NULL when the two are equal — it turns placeholders into NULL.
- `COALESCE(NULLIF(x, ''), default)` treats blanks as missing.
- Divide by `NULLIF(d, 0)` to avoid division-by-zero errors outside MySQL.

EXERCISE 1

Display names with fallbacks

The `users` table; nicknames may be NULL or left blank (`''`):

id	nickname	first_name
1	ace	Arun
2	NULL	Bela
3	(empty)	Chirag
4	NULL	NULL
5	(empty)	NULL

Return `id` and `display_name`: the nickname if it is neither NULL nor empty, otherwise the first name, otherwise `'Anonymous'`. Order by `id`.

```
id | display_name
1 | ace
2 | Bela
3 | Chirag
4 | Anonymous
5 | Anonymous
```

INPUT 1

```
CREATE TABLE users (id INT PRIMARY KEY,
nickname VARCHAR(30), first_name
VARCHAR(30));
INSERT INTO users VALUES (1, 'ace',
'Arun'), (2, NULL, 'Bela'), (3, '',
'Chirag'), (4, NULL, NULL), (5, '',
NULL);
```

OUTPUT 1

```
id | display_name
1 | ace
2 | Bela
3 | Chirag
4 | Anonymous
5 | Anonymous
```

REFERENCE SOLUTION

```sql
SELECT id,
       COALESCE(NULLIF(nickname, ''), first_name, 'Anonymous') AS display_name
FROM users
ORDER BY id;
```

EXERCISE 2

Click-through rate without dividing by zero

The `campaigns` table; `clicks` is NULL when none were recorded:

id	name	views	clicks
1	spring	200	30
2	summer	0	0
3	autumn	300	NULL
4	winter	150	50

Return `name` and `ctr` = clicks ÷ views, rounded to 2 decimal places, ordered by `id`. Treat missing clicks as 0. A campaign with 0 views (or unknown views) has no rate: its `ctr` is NULL. Guard the division with `NULLIF`.

```
name | ctr
spring | 0.15
summer | NULL
autumn | 0
winter | 0.33
```

INPUT 1

```
CREATE TABLE campaigns (id INT PRIMARY
KEY, name VARCHAR(30), views INT, clicks
INT);
INSERT INTO campaigns VALUES (1,
'spring', 200, 30), (2, 'summer', 0, 0),
(3, 'autumn', 300, NULL), (4, 'winter',
150, 50);
```

OUTPUT 1

```
name | ctr
spring | 0.15
summer | NULL
autumn | 0
winter | 0.33
```

REFERENCE SOLUTION

```sql
SELECT name,
       ROUND(COALESCE(clicks, 0) / NULLIF(views, 0), 2) AS ctr
FROM campaigns
ORDER BY id;
```

QUIZ

1. What is `COALESCE(NULL, NULL, 0, 5)`?

A. NULL

B. 0

C. 5

D. An error

2. What is `NULLIF(10, 10)`?

A. 10

B. 0

C. NULL

D. 1

3. Which is Oracle's two-argument equivalent of MySQL's `IFNULL(a, b)`?

A. `ISNULL(a, b)`

B. `NVL(a, b)`

C. `NULLIF(a, b)`

D. `IIF(a, b)`

4. Why write `total / NULLIF(count, 0)` in a query that must also run on PostgreSQL?

A. PostgreSQL raises a division-by-zero error, while dividing by NULL gives NULL

B. NULLIF makes the division faster

C. It rounds the result

D. PostgreSQL cannot divide integers otherwise

5. `SELECT salary + bonus FROM employees` returns NULL for some people. What is the usual fix when a missing bonus means no bonus?

A. `salary + NULLIF(bonus, 0)`

B. `salary + COALESCE(bonus, 0)`

C. `salary + bonus IS NOT NULL`

D. `COALESCE(salary + bonus)`

ANSWER KEY · COALESCE, IFNULL AND NULLIF

1. B COALESCE returns the first argument that is not NULL, and 0 is a value. It does not skip zeros or look further once it finds one.

2. C NULLIF returns NULL when its two arguments are equal, and the first argument otherwise; `NULLIF(10, 5)` would be 10.

3. B Oracle uses NVL, SQL Server uses ISNULL(a, b). NULLIF does the opposite job (turns a value into NULL), and IIF is SQL Server's two-way IF. COALESCE works in all of them.

4. A MySQL returns NULL for `x / 0` in a SELECT, but PostgreSQL, SQL Server and Oracle abort the query. NULLIF turns a zero divisor into NULL, so the result is NULL everywhere.

5. B COALESCE replaces the missing bonus with 0 before the addition. NULLIF would create more NULLs; a one-argument COALESCE returns the NULL sum unchanged; the IS NOT NULL version adds a 0/1 flag.

LESSON 4 · 15 MIN

CASE expressions — if-then-else in SQL

So far every column a query returned was a stored value or a formula applied the same way to every row. Real reports need decisions: a score becomes a grade band, a status code becomes a readable label, an order is "late" if it shipped after its promise date, a price gets a discount only for members. SQL makes decisions with the **CASE expression**, its if-then-else. CASE is one of the most important tools in the language — it powers custom sort orders (you already used it), conditional aggregation in the next modules, and pivot tables later — and its rules about order and NULL are simple but strict. This lesson covers the two forms of CASE, how branches are chosen, the ELSE default, where CASE can appear and MySQL's `IF()` shorthand.

Searched CASE

The general form tests a condition in each branch:

```
SELECT student, score,
       CASE
         WHEN score ⩾ 90 THEN 'A'
         WHEN score ⩾ 75 THEN 'B'
         WHEN score ⩾ 50 THEN 'C'
         ELSE 'F'
       END AS grade
FROM results;
```

student	score	grade
Asha	92	A
Ben	75	B
Dina	49	F

For each row the database evaluates the `WHEN` conditions **from top to bottom** and returns the `THEN` result of the **first one that is TRUE**. The remaining branches are not considered. If no condition is TRUE, the `ELSE` result is returned. The whole thing, from `CASE` to `END`, is one expression producing one value per row, so it gets an alias like any computed column.

First match wins

Because the first true branch decides, the order of overlapping conditions is part of the logic. Ben's 75 satisfies `score ⩾ 75` and `score ⩾ 50`; it gets `'B'` because that branch comes first. Put the branches in the wrong order and the result is quietly wrong:

```
CASE
  WHEN score ⩾ 50 THEN 'C'   -- 92 stops here
  WHEN score ⩾ 75 THEN 'B'   -- never reached for anyone ≥ 50
  WHEN score ⩾ 90 THEN 'A'
  ELSE 'F'
END
```

With ranges, go from the most restrictive condition to the least (high thresholds first when testing `⩾`), or write each band in full (`WHEN score ⩾ 75 AND score < 90`) so the order no longer matters. The first style is shorter; the second is easier to review.

ELSE, and the NULL it hides

If no branch is TRUE and there is no `ELSE`, CASE returns **NULL**. That default is convenient and dangerous: a new status code that nobody anticipated becomes a NULL label instead of an error. Write an explicit ELSE whenever any other value is possible, even `ELSE 'Unknown'`.

NULL inputs need their own branch. A condition like `score ⩾ 90` is UNKNOWN for a NULL score — not TRUE — so a NULL falls through every comparison to the ELSE. In the grade query above, a student with no score would get `'F'`, which is a lie. Test for NULL **first**:

```
CASE
  WHEN score IS NULL THEN 'absent'
  WHEN score ⩾ 90 THEN 'A'
  WHEN score ⩾ 75 THEN 'B'
  WHEN score ⩾ 50 THEN 'C'
  ELSE 'F'
END AS grade
```

Simple CASE

When every branch compares the same expression with a constant, the **simple** form is shorter:

```
SELECT id,
       CASE status_code
         WHEN 'P' THEN 'Pending'
         WHEN 'S' THEN 'Shipped'
         WHEN 'D' THEN 'Delivered'
         WHEN 'C' THEN 'Cancelled'
         ELSE 'Unknown'
       END AS status
FROM orders;
```

`CASE status_code WHEN 'P' …` means `CASE WHEN status_code = 'P' …`. It uses `=`, so the three-valued rules apply: `WHEN NULL` can never match (since `NULL = NULL` is UNKNOWN), and a NULL `status_code` falls through to the ELSE. If NULL needs its own label, switch to the searched form with `WHEN status_code IS NULL`. Simple CASE can only test equality; anything with `<`, `LIKE`, `IN`, `AND` or `IS` needs the searched form.

The result type

All branches of a CASE should produce the same type. Mixing them — `THEN 1` in one branch, `THEN 'none'` in another — makes the database pick a common type, usually a string in MySQL, so the column stops sorting numerically. PostgreSQL refuses outright to mix integer and text branches. Keep branch results consistent, and give every branch a result even if it is NULL (`THEN NULL` is allowed).

CASE anywhere a value is allowed

CASE is an expression, not a statement, so it can go anywhere a value can:

```sql
-- In ORDER BY: a custom order (as in the sorting module)
ORDER BY CASE status_code WHEN 'P' THEN 1 WHEN 'S' THEN 2 ELSE 3 END, id

-- In WHERE: a condition that depends on another column
WHERE amount > CASE WHEN customer_type = 'business' THEN 10000 ELSE 2000 END

-- Inside arithmetic: a member discount
SELECT price * CASE WHEN is_member = 1 THEN 0.9 ELSE 1 END AS charged
FROM orders;

-- Inside another function
SELECT COALESCE(CASE WHEN qty > 0 THEN 'in stock' END, 'sold out') AS availab
FROM products;
```

One restriction catches people: a SELECT alias cannot be used in another column of the same SELECT, nor in WHERE. If two columns depend on the same CASE, repeat the expression (or, later, compute it once in a subquery or CTE). In ORDER BY you can refer to the alias directly, since ORDER BY runs after SELECT — `ORDER BY grade` works.

In the coming modules CASE gets even more useful inside aggregates: `SUM(CASE WHEN status = 'paid' THEN amount ELSE 0 END)` totals only the paid orders in a single pass, the basis of pivot-style reports.

MySQL's IF()

For a single condition with two outcomes MySQL has a function shorthand:

```sql
SELECT name, IF(stock > 0, 'in stock', 'sold out') AS availability
FROM products;
```

`IF(cond, a, b)` returns `a` when the condition is TRUE and `b` when it is FALSE **or UNKNOWN** — so a NULL stock gives `'sold out'`, the same as `CASE WHEN stock > 0 THEN … ELSE … END`. IF is MySQL-only (SQL Server's `IIF` is similar); prefer CASE in answers that should be portable, and do not confuse `IF()` the function with the `IF … THEN … END IF` statement of stored procedures.

Before you move on

- Searched CASE: the first `WHEN` that is TRUE wins; order overlapping conditions carefully.
- No ELSE means `ELSE NULL`; write one whenever other values are possible.
- NULL inputs fail every comparison, so test `IS NULL` first.
- Simple CASE compares with `=`: `WHEN NULL` never matches.
- CASE is an expression: SELECT, WHERE, ORDER BY, inside functions; `IF(c, a, b)` is MySQL's two-way shorthand.

EXERCISE 1

Grade bands

The `results` table; `score` is NULL for students who missed the exam:

id	student	score
1	Asha	92
2	Ben	75
3	Chen	NULL
4	Dina	49
5	Eli	50
6	Fay	89

Return `student`, `score` and `grade`, ordered by `id`: `'absent'` when the score is NULL, `'A'` for 90 and above, `'B'` for 75–89, `'C'` for 50–74, `'F'` below 50.

```
student | score | grade
Asha | 92 | A
Ben | 75 | B
Chen | NULL | absent
Dina | 49 | F
Eli | 50 | C
Fay | 89 | B
```

INPUT 1

```
CREATE TABLE results (id INT PRIMARY
KEY, student VARCHAR(40), score INT);
INSERT INTO results VALUES (1, 'Asha',
92), (2, 'Ben', 75), (3, 'Chen', NULL),
(4, 'Dina', 49), (5, 'Eli', 50), (6,
'Fay', 89);
```

OUTPUT 1

```
student | score | grade
Asha | 92 | A
Ben | 75 | B
Chen | NULL | absent
Dina | 49 | F
Eli | 50 | C
Fay | 89 | B
```

REFERENCE SOLUTION

```sql
SELECT student,
       score,
       CASE
         WHEN score IS NULL THEN 'absent'
         WHEN score ⩾ 90 THEN 'A'
         WHEN score ⩾ 75 THEN 'B'
         WHEN score ⩾ 50 THEN 'C'
         ELSE 'F'
       END AS grade
FROM results
ORDER BY id;
```

EXERCISE 2

Order status board

The `orders` table stores a one-letter status code:

id	status_code
1	D
2	P
3	C
4	S
5	P
6	X
7	NULL

Return `id` and `status`, where P → `Pending`, S → `Shipped`, D → `Delivered`, C → `Cancelled` and anything else (including NULL) → `Unknown`. Order the board Pending, Shipped, Delivered, Cancelled, Unknown, and by `id` within each status.

```
id | status
2 | Pending
5 | Pending
4 | Shipped
1 | Delivered
3 | Cancelled
6 | Unknown
7 | Unknown
```

INPUT 1

```
CREATE TABLE orders (id INT PRIMARY KEY,
status_code CHAR(1));
INSERT INTO orders VALUES (1, 'D'), (2,
'P'), (3, 'C'), (4, 'S'), (5, 'P'), (6,
'X'), (7, NULL);
```

OUTPUT 1

```
id | status
2 | Pending
5 | Pending
4 | Shipped
1 | Delivered
3 | Cancelled
6 | Unknown
7 | Unknown
```

REFERENCE SOLUTION

```sql
SELECT id,
       CASE status_code
         WHEN 'P' THEN 'Pending'
         WHEN 'S' THEN 'Shipped'
         WHEN 'D' THEN 'Delivered'
         WHEN 'C' THEN 'Cancelled'
         ELSE 'Unknown'
       END AS status
FROM orders
ORDER BY CASE status_code
           WHEN 'P' THEN 1
           WHEN 'S' THEN 2
           WHEN 'D' THEN 3
           WHEN 'C' THEN 4
           ELSE 5
         END,
         id;
```

QUIZ

1. What grade does a score of 95 get?

```
CASE
  WHEN score ⩾ 50 THEN 'C'
  WHEN score ⩾ 75 THEN 'B'
  WHEN score ⩾ 90 THEN 'A'
END
```

A. A

B. B

C. C

D. NULL

2. What does `CASE WHEN qty > 10 THEN 'bulk' END` return for `qty = 3`?

A. 'bulk'

B. An empty string

C. NULL

D. An error, because ELSE is missing

3. For a row where `code` is NULL, what does `CASE code WHEN NULL THEN 'none' ELSE 'some' END` return?

A. 'none'

B. 'some'

C. NULL

D. An error

4. What is `IF(NULL > 1, 'yes', 'no')` in MySQL?

A. 'yes'

B. 'no'

C. NULL

D. An error

5. Where can a CASE expression NOT be used?

A. In the SELECT list

B. In WHERE

C. In ORDER BY

D. As a table name in FROM

ANSWER KEY · CASE EXPRESSIONS — IF-THEN-ELSE IN SQL

1. C CASE returns the first branch that is TRUE, and `95 ⩾ 50` is checked first. Order range tests from the highest threshold down.

2. C A CASE without ELSE has an implicit `ELSE NULL`. It is not an error; that is exactly why unexpected values silently become NULL.

3. B A simple CASE compares with `=`, and `NULL = NULL` is UNKNOWN, so the WHEN never matches and the ELSE wins. Use the searched form `CASE WHEN code IS NULL THEN 'none' …`.

4. B IF returns its second argument only when the condition is TRUE; for FALSE and UNKNOWN it returns the third. `CASE WHEN … THEN 'yes' ELSE 'no' END` behaves the same.

5. D CASE is a value-producing expression, allowed anywhere a value is — SELECT, WHERE, ORDER BY, inside functions and aggregates. It cannot name a table.

LESSON 5 · 15 MIN

Data types, implicit conversion and CAST

Every value in SQL has a type: an integer, an exact decimal, a string, a date. Most of the time you do not think about it, because the database converts between types for you — compare a number column with `'100'` and MySQL quietly treats the string as a number. Those quiet conversions are where surprises live: numbers stored as text sort as text, `'abc' = 0` is true in MySQL, `7 / 2` gives `3.5` in MySQL but `3` in PostgreSQL. This lesson covers the main MySQL types, the rules of implicit conversion, integer and decimal arithmetic, and explicit conversion with `CAST` and `CONVERT`. Choosing types for a new table is covered in the schema-design module; here the focus is what happens to values inside a query.

The main MySQL types

Family	Types	Notes
Integer	`TINYINT`, `SMALLINT`, `INT`, `BIGINT`	`INT` is 32-bit (±2.1 billion); `BIGINT` for ids that can grow past that
Exact decimal	`DECIMAL(p, s)`	`p` digits in all, `s` after the point: `DECIMAL(10,2)` holds up to 99,999,999.99
Floating point	`FLOAT`, `DOUBLE`	Approximate binary fractions
Text	`CHAR(n)`, `VARCHAR(n)`, `TEXT`	`CHAR` pads to a fixed length, `VARCHAR` stores what it is given
Date and time	`DATE`, `DATETIME`, `TIMESTAMP`, `TIME`	Literals are written `'2024-05-03'` and `'2024-05-03 14:30:00'`
Boolean	`BOOLEAN`	An alias for `TINYINT(1)`: `TRUE` is 1, `FALSE` is 0

That last row explains something you have already used: in MySQL a comparison like `price > 100` *is* a number, 1 or 0, which is why it can be selected, summed and sorted on.

DECIMAL versus FLOAT

Floating-point types store binary fractions, and most decimal fractions have no exact binary form — the same reason `0.1 + 0.2` is not exactly `0.3` in JavaScript or Python. MySQL's `DOUBLE` arithmetic carries the same tiny errors, and summing thousands of such values drifts. `DECIMAL` stores decimal digits exactly. The rule is simple and interviewers expect it: **money and anything that must add up exactly goes in `DECIMAL`**; `FLOAT` / `DOUBLE` are for measurements where a tiny relative error is fine.

Arithmetic: `/` and DIV

MySQL's division operator always produces a decimal result:

```
SELECT 7 / 2    AS a,   -- 3.5000
       6 / 3    AS b,   -- 2.0000
       7 DIV 2  AS c,   -- 3   (integer division, truncated toward zero)
       7 % 2    AS d,   -- 1   (remainder; MOD(7, 2) is the same)
       -7 DIV 2 AS e;   -- -3
```

MySQL shows four decimal places for `/` by default (`div_precision_increment`). For integer division use `DIV`, and for the remainder `%` or `MOD()`. Division by zero gives NULL in a MySQL SELECT, as the previous lesson noted.

Other databases differ in exactly the place that causes bugs: in **PostgreSQL, SQL Server and SQLite, an integer divided by an integer is an integer**. `7 / 2` is `3` there, and `SUM(passed) / COUNT(*)` is `0` for any pass rate below 100%. The portable fix is to make one side non-integer — `SUM(passed) * 1.0 / COUNT(*)` or a `CAST(… AS DECIMAL(10,4))` — which is harmless in MySQL and correct everywhere.

Implicit conversion

When an operator gets two different types, MySQL converts one of them. The rules worth knowing:

- **Arithmetic converts strings to numbers.** `'10' + 5` is `15`. A string that does not start with a number becomes `0` (`'abc' + 1` is `1`), and one with a numeric prefix keeps the prefix (`'12abc' + 0` is `12`), each with a warning.
- **A comparison between a string and a number compares numerically.** `'10' = 10` is true, `'abc' = 0` is true, and `'1e1' = 10` is true. This is MySQL-specific — PostgreSQL refuses to

compare text with an integer without a cast, and SQLite compares a literal number with literal text by type (any number is less than any text).

- **A comparison between two strings compares as strings**, by collation: `'10' < '9'` is true.
- **Dates compare correctly with `'YYYY-MM-DD'` strings**, because MySQL converts the string to a date: `WHERE order_date ⩾ '2024-01-01'` is the normal way to write a date filter.

Implicit conversion is convenient in quick queries and risky in real ones. Comparing an indexed `VARCHAR` column with a number, for example `WHERE phone = 9876543210`, forces MySQL to convert every phone to a number, which defeats the index and matches `'09876543210'` too. Match the column's type in your literals: `WHERE phone = '9876543210'`.

CAST and CONVERT

When you need a conversion, say so. The standard form is `CAST(expr AS type)`; MySQL also accepts `CONVERT(expr, type)`:

```sql
SELECT CAST('42' AS SIGNED)           AS n,    -- 42
       CAST('007' AS UNSIGNED)        AS m,    -- 7
       CAST(42 AS CHAR)               AS s,    -- '42'
       CAST('2024-05-03' AS DATE)     AS d,    -- 2024-05-03
       CAST(1234.5678 AS DECIMAL(6,2)) AS x,   -- 1234.57
       CONVERT('15', SIGNED)          AS y;    -- 15
```

MySQL's integer target types are `SIGNED` and `UNSIGNED` (optionally followed by `INTEGER`) — `CAST(x AS INT)`, which PostgreSQL and SQL Server accept, is a syntax error in MySQL. Other targets: `DECIMAL(p, s)`, `CHAR` or `CHAR(n)`, `DATE`, `DATETIME`, `TIME`, `DOUBLE`, `FLOAT` and `JSON`. You cannot cast to `VARCHAR` in MySQL — use `CHAR`.

Some details differ by database and appear in quizzes. Casting a fractional number to an integer **rounds** in MySQL (`CAST(3.7 AS SIGNED)` is 4), while SQLite and `int()`-style conversions in programming languages **truncate**; casting the *string* `'3.7'` in MySQL keeps only the integer prefix, 3. SQL Server's `CONVERT(type, expr)` takes its arguments in the opposite order to MySQL's, and adds a style number for date formats.

The everyday uses

Sorting numbers stored as text. Ticket codes, house numbers and imported spreadsheet columns often arrive as `VARCHAR`. Sort them by value:

```sql
SELECT code
FROM tickets
ORDER BY CAST(code AS UNSIGNED), id;
```

Forcing decimal division in code that must run on several databases: `CAST(passed AS DECIMAL(10,4)) / total`.

Comparing a text column numerically, when you cannot change the schema: `WHERE CAST(zip AS UNSIGNED) BETWEEN 400001 AND 400099`. It works, but it cannot use an index on `zip` — the real fix is the right column type.

Before you move on

- `DECIMAL` is exact; `FLOAT` / `DOUBLE` are approximate — money goes in `DECIMAL`.
- MySQL's `/` is decimal division; `DIV` is integer division, `%` / `MOD` the remainder; other databases truncate `int / int`.
- MySQL converts strings to numbers in arithmetic and in string-to-number comparisons; two strings compare as text.
- `CAST(x AS SIGNED | UNSIGNED | DECIMAL(p,s) | CHAR | DATE)`; MySQL also has `CONVERT(x, type)`.
- Match literal types to column types to avoid silent conversions.

EXERCISE 1

Ticket codes sorted as numbers

The `tickets` table stores its codes as text (`VARCHAR`), some with leading zeros:

id	code
1	9
2	10
3	100
4	007
5	25

Return `code` and `code_num` (the code converted to an integer with `CAST`), ordered by the **numeric** value, then by `id`.

```
code | code_num
007 | 7
9 | 9
10 | 10
25 | 25
100 | 100
```

INPUT 1

```
CREATE TABLE tickets (id INT PRIMARY
KEY, code VARCHAR(10));
INSERT INTO tickets VALUES (1, '9'), (2,
'10'), (3, '100'), (4, '007'), (5,
'25');
```

OUTPUT 1

```
code | code_num
007 | 7
9 | 9
10 | 10
25 | 25
100 | 100
```

REFERENCE SOLUTION

```
SELECT code,
       CAST(code AS UNSIGNED) AS code_num
FROM tickets
ORDER BY CAST(code AS UNSIGNED), id;
```

EXERCISE 2

Packing boxes

The `shipments` table: how many `items` each shipment has and how many fit in one box (`per_box`, NULL when not decided yet):

id	items	per_box
1	25	6
2	24	6
3	5	12
4	0	4

Return `id`, `full_boxes` (integer division with `DIV`), `left_over` (the remainder) and `boxes_needed` (full boxes, plus one more if anything is left over), ordered by `id`. When `per_box` is NULL every

computed column is NULL.

```
id | full_boxes | left_over | boxes_needed
1 | 4 | 1 | 5
2 | 4 | 0 | 4
3 | 0 | 5 | 1
4 | 0 | 0 | 0
```

INPUT 1

```
CREATE TABLE shipments (id INT PRIMARY
KEY, items INT, per_box INT);
INSERT INTO shipments VALUES (1, 25, 6),
(2, 24, 6), (3, 5, 12), (4, 0, 4);
```

OUTPUT 1

```
id | full_boxes | left_over |
boxes_needed
1 | 4 | 1 | 5
2 | 4 | 0 | 4
3 | 0 | 5 | 1
4 | 0 | 0 | 0
```

REFERENCE SOLUTION

```sql
SELECT id,
       items DIV per_box AS full_boxes,
       items % per_box AS left_over,
       items DIV per_box + (items % per_box > 0) AS boxes_needed
FROM shipments
ORDER BY id;
```

QUIZ

1. What is `7 / 2` in MySQL, and in PostgreSQL (both operands integers)?

A. 3.5000 in both

B. 3.5000 in MySQL, 3 in PostgreSQL

C. 3 in both

D. 3 in MySQL, 3.5 in PostgreSQL

2. In MySQL, what is `'10' < '9'`?

A. 1 (true)

B. 0 (false)

C. NULL

D. An error

3. Which is a valid MySQL cast to an integer?

A. `CAST(x AS INT)`

B. `CAST(x AS SIGNED)`

C. `CAST(x AS NUMBER)`

D. `CAST(x AS VARCHAR)`

4. Which type should store prices in rupees and paise?

A. `FLOAT`

B. `DOUBLE`

C. `DECIMAL(10,2)`

D. `VARCHAR(10)`

5. What does MySQL return for `'abc' = 0`?

A. 0

B. 1

C. NULL

D. An error

ANSWER KEY · DATA TYPES, IMPLICIT CONVERSION AND CAST

1. B MySQL's `/` always does decimal division; PostgreSQL, SQL Server and SQLite truncate integer ÷ integer. Use `DIV` in MySQL for integer division, and multiply by 1.0 or CAST for decimal division everywhere.

2. A Two strings compare as strings, character by character, and `'1'` < `'9'`. If one side were the number 9, MySQL would compare numerically and the answer would be 0.

3. B MySQL's integer targets are SIGNED and UNSIGNED. `AS INT` works in PostgreSQL and SQL Server but not MySQL; NUMBER is Oracle; and MySQL casts to CHAR, not VARCHAR.

4. C DECIMAL stores exact decimal digits, so sums never drift. FLOAT and DOUBLE are binary approximations (0.1 + 0.2 is not exactly 0.3), and text cannot be summed or compared numerically without conversion.

5. B Comparing a string with a number converts the string to a number, and a string with no numeric prefix becomes 0 — so the comparison is true (with a warning). PostgreSQL would refuse the comparison.

CHECKPOINT · 25 MIN

Checkpoint — NULLs and expressions

This checkpoint covers the whole module: what NULL means, how it spreads through arithmetic and comparisons, three-valued logic in `WHERE`, `AND`, `OR`, `NOT` and `NOT IN`, `IS NULL` and `IS NOT NULL`, MySQL's NULL-safe `<=>`, `COALESCE`, `IFNULL` and `NULLIF`, searched and simple `CASE`, MySQL's `IF()`, the main data types, `/` versus `DIV`, implicit conversion, and `CAST` / `CONVERT`.

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 are `NULL = NULL`, `NULL + 1` and `NULL OR TRUE`?
- Why does `WHERE col <> 'x'` drop NULL rows, and how do you keep them?
- Why does `NOT IN (1, 2, NULL)` never match anything?
- What is the difference between `IS NULL`, `= NULL` and `<=>`?
- What do `COALESCE`, `IFNULL` and `NULLIF` return, and what are SQL Server's and Oracle's names for `IFNULL`?
- In a searched `CASE`, which branch wins, and what is returned when none matches?
- What is `7 / 2` in MySQL, and in PostgreSQL? What is `7 DIV 2`?
- How do you sort a `VARCHAR` column of numbers by value?

The three queries are a delivery fee computed with `CASE` and reduced by a discount that may be missing, a contact list that treats blank cities as unknown and lists reachable contacts first, and an audit that labels each change as added, removed, changed or the same.

EXERCISE 1

Delivery fees with optional discounts

The `deliveries` table; `weight_kg` is NULL when the parcel was not weighed and `discount` is NULL when there is none:

id	city	weight_kg	discount
1	Pune	0.5	NULL
2	Delhi	3	10
3	Goa	NULL	5
4	Agra	12	25
5	Kochi	1	0

Return `id`, `fee` and `final_fee`, ordered by `id`. The `fee` is 40 up to 1 kg, 80 up to 5 kg, 150 above 5 kg, and NULL for an unweighed parcel. `final_fee` is the fee minus the discount, a missing discount counting as 0.

```
id | fee | final_fee
1 | 40 | 40
2 | 80 | 70
3 | NULL | NULL
4 | 150 | 125
5 | 40 | 40
```

INPUT 1

```
CREATE TABLE deliveries (id INT PRIMARY
KEY, city VARCHAR(30), weight_kg
DECIMAL(5,2), discount INT);
INSERT INTO deliveries VALUES (1,
'Pune', 0.5, NULL), (2, 'Delhi', 3, 10),
(3, 'Goa', NULL, 5), (4, 'Agra', 12,
25), (5, 'Kochi', 1, 0);
```

OUTPUT 1

```
id | fee | final_fee
1 | 40 | 40
2 | 80 | 70
3 | NULL | NULL
4 | 150 | 125
5 | 40 | 40
```

REFERENCE SOLUTION

```sql
SELECT id,
       CASE
         WHEN weight_kg IS NULL THEN NULL
         WHEN weight_kg ⩽ 1 THEN 40
         WHEN weight_kg ⩽ 5 THEN 80
         ELSE 150
       END AS fee,
       CASE
         WHEN weight_kg IS NULL THEN NULL
         WHEN weight_kg ⩽ 1 THEN 40
         WHEN weight_kg ⩽ 5 THEN 80
         ELSE 150
       END - COALESCE(discount, 0) AS final_fee
FROM deliveries
ORDER BY id;
```

EXERCISE 2

Contact list, reachable first

The `contacts` table; `city` and `phone` may be NULL or left blank (`''`):

id	name	city	phone
1	Ravi	Pune	9876
2	Anu	(empty)	NULL
3	Mo	NULL	9123
4	Lia	Goa	NULL

Return `name`, `city` and `has_phone`: the city, or `'Unknown'` when it is NULL or blank; and 1 if the contact has a phone that is neither NULL nor blank, else 0. List contacts with a phone first, then by `name`, then by `id`.

```
name | city | has_phone
Mo | Unknown | 1
Ravi | Pune | 1
Anu | Unknown | 0
Lia | Goa | 0
```

INPUT 1

```
CREATE TABLE contacts (id INT PRIMARY
KEY, name VARCHAR(30), city VARCHAR(30),
phone VARCHAR(20));
INSERT INTO contacts VALUES (1, 'Ravi',
'Pune', '9876'), (2, 'Anu', '', NULL),
(3, 'Mo', NULL, '9123'), (4, 'Lia',
'Goa', NULL);
```

OUTPUT 1

```
name | city | has_phone
Mo | Unknown | 1
Ravi | Pune | 1
Anu | Unknown | 0
Lia | Goa | 0
```

REFERENCE SOLUTION

```sql
SELECT name,
       COALESCE(NULLIF(city, ''), 'Unknown') AS city,
       NULLIF(phone, '') IS NOT NULL AS has_phone
FROM contacts
ORDER BY has_phone DESC, name, id;
```

EXERCISE 3

Audit labels

The `audit` table holds a field's value before and after an edit; NULL means the field was empty:

id	before_value	after_value
1	NULL	5
2	5	NULL
3	5	6
4	5	5
5	NULL	NULL

Return `id` and `change_type`, ordered by `id`: `'added'` when a value appeared, `'removed'` when it disappeared, `'changed'` when both exist and differ, and `'same'` otherwise (including NULL to NULL).

```
id | change_type
1 | added
2 | removed
3 | changed
4 | same
5 | same
```

INPUT 1

```
CREATE TABLE audit (id INT PRIMARY KEY,
before_value INT, after_value INT);
INSERT INTO audit VALUES (1, NULL, 5),
(2, 5, NULL), (3, 5, 6), (4, 5, 5), (5,
NULL, NULL);
```

OUTPUT 1

```
id | change_type
1 | added
2 | removed
3 | changed
4 | same
5 | same
```

REFERENCE SOLUTION

```sql
SELECT id,
       CASE
         WHEN before_value IS NULL AND after_value IS NOT NULL THEN 'added'
         WHEN before_value IS NOT NULL AND after_value IS NULL THEN 'removed'
         WHEN before_value <> after_value THEN 'changed'
         ELSE 'same'
       END AS change_type
FROM audit
ORDER BY id;
```

CHECKPOINT QUESTIONS

1. What is `NULL OR TRUE`?

A. TRUE

B. FALSE

C. UNKNOWN

D. NULL is not allowed in OR

2. What does `SELECT 10 + NULL` return?

A. 10

B. 0

C. NULL

D. An error

3. Which condition is TRUE exactly for rows where `email` has no value?

A. `email = NULL`

B. `email = ''`

C. `email IS NULL`

D. `NOT email`

4. What does MySQL return for `NULL ⟺ NULL`?

A. NULL

B. 0

C. 1

D. An error

5. A query must keep every row where `NOT (discount > 10)` — including rows whose discount is NULL. Which works?

A. `WHERE NOT (discount > 10)`

B. `WHERE discount ⩽ 10`

C. `WHERE discount ⩽ 10 OR discount IS NULL`

D. `WHERE discount ≠ 10`

6. What is `COALESCE(NULLIF('', ''), 'n/a')`?

A. ''

B. 'n/a'

C. NULL

D. An error

7. Which pair are both SQL Server functions?

A. `ISNULL(a, b)` and `IIF(c, a, b)`

B. `IFNULL(a, b)` and `IF(c, a, b)`

C. `NVL(a, b)` and `DECODE`

D. `ISNULL(x)` and `⟺`

8. What does this return for `qty = NULL`?

```sql
CASE WHEN qty > 0 THEN 'in stock' ELSE 'sold out' END
```

A. 'in stock'

B. 'sold out'

C. NULL

D. An error

9. What is `CASE 'x' WHEN 'a' THEN 1 WHEN 'b' THEN 2 END`?

A. 0

B. NULL

C. An error

D. 3

10. What is `17 DIV 5` and `17 % 5` in MySQL?

A. 3.4 and 2

B. 3 and 2

C. 3 and 3

D. 4 and 2

11. A `VARCHAR` column holds `'2'`, `'10'` and `'1'`. Which ORDER BY gives 1, 2, 10?

A. `ORDER BY code`

B. `ORDER BY code DESC`

C. `ORDER BY CAST(code AS UNSIGNED)`

D. `ORDER BY CAST(code AS CHAR)`

12. Why is `WHERE phone = 9876543210` risky when `phone` is a `VARCHAR` column in MySQL?

A. It is a syntax error

B. MySQL converts every phone to a number to compare, which skips the index and also matches values like '09876543210'

C. It compares only the first character

D. It always returns no rows

ANSWER KEY · CHECKPOINT — NULLS AND EXPRESSIONS

1. A OR is true as soon as one side is true, so the unknown side cannot change the answer. `NULL OR FALSE` would be UNKNOWN.

2. C Arithmetic with an unknown value is unknown. Use `10 + COALESCE(x, 0)` when a missing value should count as zero.

3. C `= NULL` is UNKNOWN for every row, `= ''` finds empty strings (a value), and `NOT email` converts the text to a number. IS NULL is the dedicated test.

4. C `⟺` is NULL-safe equality: two NULLs count as equal, giving 1. With `=` the result would be NULL (UNKNOWN).

5. C For a NULL discount `discount > 10` is UNKNOWN, and NOT UNKNOWN is still UNKNOWN, so the first two drop those rows. The explicit IS NULL branch keeps them. (`(discount > 10) IS NOT TRUE` also works in MySQL.)

6. B NULLIF turns the empty string into NULL because the two arguments are equal, and COALESCE skips that NULL to return 'n/a'.

7. A SQL Server has ISNULL(a, b) and IIF. IFNULL and IF() are MySQL, NVL and DECODE are Oracle, and one-argument ISNULL(x) and `⟺` are MySQL. COALESCE and CASE are standard everywhere.

8. B `NULL > 0` is UNKNOWN, not TRUE, so the WHEN is skipped and the ELSE is returned. If unknown stock should read differently, add `WHEN qty IS NULL THEN 'unknown'` first.

9. B No WHEN matches and there is no ELSE, so the implicit `ELSE NULL` applies.

10. B DIV is integer division (17 / 5 = 3.4, truncated to 3) and % is the remainder (17 − 15 = 2). `17 / 5` would give 3.4000.

11. C As text the order is '1', '10', '2'. Casting to an integer sorts by value. Casting to CHAR keeps it text.

12. B A string-to-number comparison is done numerically, so each stored string is converted — the index on the text cannot be used and leading zeros disappear. Quote the literal: `phone = '9876543210'`.

CodeKairo · SQL study plan · Module 4: NULLs and expressions. Solve the exercises online to track your progress and earn XP.