Module 5 of 16

Functions

String functions to join, clean, cut and pad text; ROUND, CEIL, FLOOR, DIV and MOD; date types, date parts and DATE_FORMAT; date arithmetic with INTERVAL, DATEDIFF and TIMESTAMPDIFF; and IF, IFNULL, NULLIF, GREATEST and FIELD.

Stored data is rarely in the shape a report wants. Names arrive with stray spaces and in the wrong case, emails need splitting into a user and a domain, prices need tax and two decimals, dates need to become month keys and due dates, and a status needs a label and a sort order that is not alphabetical. MySQL's built-in functions do that reshaping inside the query, row by row, without changing a stored value. This module teaches the functions that appear in nearly every real report and interview answer, the rules they share — positions start at 1, NULL in means NULL out — and where PostgreSQL and SQL Server spell them differently.

String functions covers `CONCAT` and `CONCAT_WS`, case and `CHAR_LENGTH`, `TRIM`, `SUBSTRING`, `LEFT` / `RIGHT`, `LOCATE`, `REPLACE`, `SUBSTRING_INDEX` and `LPAD`. Numeric functions covers MySQL's decimal `/`, `DIV` and `MOD`, `ROUND`, `CEIL`, `FLOOR` and `TRUNCATE`, `ABS` and the money types. Dates and times covers the types, `YEAR` / `MONTH` / `DAYNAME`, the two weekday numberings, `DATE_FORMAT` and filtering a period with a half-open range. Date arithmetic covers `DATE_ADD` with `INTERVAL`, month-end clamping, `DATEDIFF`, `TIMESTAMPDIFF` and `LAST_DAY`. Conditional functions covers `IF`, `IFNULL`, `NULLIF` for safe division, `GREATEST` / `LEAST` and `FIELD` for custom sort orders.

The exercises are single-table queries on small, untidy datasets: display names and initials, email parts and masks, prices with tax, boxes and leftovers, month keys and weekend orders, loan due dates and late days, subscription renewals and tenure, a stock board in a custom order and click-through rates that survive zero views. The checkpoint adds a contact-card formatter, an invoice report with late fees and a carrier scorecard.

CONTENTS

1	String functions — CONCAT, SUBSTRING, TRIM and friends	14 min
2	Numeric functions — ROUND, CEIL, FLOOR, MOD and division	13 min
3	Dates and times — types, parts and DATE_FORMAT	14 min
4	Date arithmetic — INTERVAL, DATEDIFF and TIMESTAMPDIFF	14 min

5	Conditional functions — IF, IFNULL, NULLIF, GREATEST and FIELD	13 min
✓	Checkpoint — Functions	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 · 14 MIN

String functions — CONCAT, SUBSTRING, TRIM and friends

Most of the text in a database was typed by people, and it arrives the way people type: names in shouting capitals, stray spaces around an email, phone numbers with and without dashes, a full name in one column when the report wants it in two. String functions are how a query cleans and reshapes that text on the way out, without changing a single stored row. This lesson covers the functions MySQL programmers use every day, the rules they share, and the places where other databases spell them differently.

The shared rules

Three rules hold for nearly every string function, and knowing them saves a lot of surprises:

- **Positions start at 1.** The first character of `'hello'` is at position 1, not 0.
- **NULL in, NULL out.** Pass NULL to `UPPER`, `SUBSTRING`, `REPLACE` or `CONCAT` and the result is NULL. `CONCAT_WS` is the notable exception.
- **Functions return new values.** `UPPER(name)` in a SELECT changes what is printed, never what is stored. Changing data is `UPDATE`'s job, in a later module.

```
SELECT UPPER('sql'), SUBSTRING('hello', 1, 2), UPPER(NULL);
```

UPPER('sql')	SUBSTRING('hello', 1, 2)	UPPER(NULL)
SQL	he	NULL

Joining text: CONCAT and CONCAT_WS

`CONCAT` glues any number of arguments together. Numbers are turned into text on the way.

```
SELECT CONCAT(first_name, ' ', last_name) AS full_name,
       CONCAT('#', order_id)              AS ref
FROM customers;
```

The catch is NULL: if any argument is NULL, the whole result is NULL. A customer with no middle name makes `CONCAT(first, ' ', middle, ' ', last)` vanish entirely. `CONCAT_WS` — "with separator" — takes the separator first, puts it between the arguments and **skips** NULL arguments:

```
SELECT CONCAT_WS(' ', 'Ada', NULL, 'Lovelace');   -- 'Ada Lovelace'
SELECT CONCAT(' ', 'Ada', NULL, 'Lovelace');      -- NULL
SELECT CONCAT_WS(', ', city, state, country)      -- 'Pune, India' when state
FROM addresses;
```

In PostgreSQL, SQLite and standard SQL the concatenation operator is `||`. In MySQL, `||` is logical OR by default, so `'a' || 'b'` is `0` — a classic surprise for people moving between databases. Write `CONCAT` in MySQL.

Changing case and measuring

`UPPER` and `LOWER` (also spelled `UCASE` / `LCASE`) change the case of every letter. `CHAR_LENGTH` counts characters.

```
SELECT LOWER(email)       AS email,
       CHAR_LENGTH(name)  AS letters
FROM users;
```

`LENGTH` also exists, but it counts **bytes**. For ASCII the two agree; for `'café'` in UTF-8, `LENGTH` is 5 and `CHAR_LENGTH` is 4. When a question says "names longer than 10 characters", it means `CHAR_LENGTH`. SQL Server calls its character count `LEN` (which also ignores trailing spaces), and PostgreSQL's `LENGTH` counts characters.

Cleaning: TRIM, LTRIM, RTRIM

`TRIM` removes spaces from both ends; `LTRIM` and `RTRIM` from one end. Spaces inside the text stay.

```
SELECT CONCAT('[', TRIM('  Ada Lovelace  '), ']');   -- '[Ada Lovelace]'
SELECT CONCAT('[', LTRIM('  Ada  '), ']');           -- '[Ada  ]'
```

MySQL's `TRIM` also takes a form that strips other characters — `TRIM(LEADING '0' FROM '00042')` gives `'42'`, and `TRIM(BOTH '-' FROM '--x--')` gives `'x'`. Trimming is the first step in comparing user-typed values: `' ada@x.com'` and `'ada@x.com'` are different strings until you trim them.

Cutting pieces out

`SUBSTRING(s, start, length)` returns `length` characters starting at `start`. Leave out the length to take everything to the end; give a negative start to count from the right. `SUBSTR` and `MID` are synonyms.

```
SELECT SUBSTRING('2024-03-15', 1, 4)  AS yr,     -- '2024'
       SUBSTRING('2024-03-15', 6, 2)  AS mon,    -- '03'
       SUBSTRING('database', 5)       AS rest,   -- 'base'
       SUBSTRING('database', -4)      AS tail;   -- 'base'
```

`LEFT(s, n)` and `RIGHT(s, n)` take the first or last `n` characters. They read better than `SUBSTRING` when that is all you want — masking a card number is `CONCAT('**** ', RIGHT(card, 4))`.

Finding and replacing

`LOCATE(sub, s)` returns the position of the first `sub` inside `s`, or **0** when it is not there (0 is never a valid position, which is what makes it a safe "not found"). `INSTR(s, sub)` is the same with the arguments the other way round, and `POSITION(sub IN s)` is the standard spelling.

```
SELECT LOCATE('@', 'ada@example.com');                       -- 4
SELECT SUBSTRING('ada@example.com', LOCATE('@', 'ada@example.com') + 1);   --
```

`REPLACE(s, from, to)` swaps **every** occurrence, case-sensitively:

```
SELECT REPLACE('98-765-4321', '-', '');     -- '987654321'
SELECT REPLACE('a.b.c', '.', '/');          -- 'a/b/c'
```

Chaining `REPLACE` calls is the usual way to strip several characters: `REPLACE(REPLACE(phone, '-', ''), ' ', '')`.

Splitting on a delimiter: SUBSTRING_INDEX

`SUBSTRING_INDEX(s, delim, n)` is MySQL's most useful splitting tool. With a positive `n` it returns everything **before** the n-th delimiter; with a negative `n`, everything **after** the n-th delimiter counting from the right.

```
SELECT SUBSTRING_INDEX('ada@example.com', '@', 1)    AS user_part,   -- 'ada'
       SUBSTRING_INDEX('ada@example.com', '@', -1)   AS domain,      -- 'examp
       SUBSTRING_INDEX('www.mysql.com', '.', 2)      AS first_two,   -- 'www.m
       SUBSTRING_INDEX('www.mysql.com', '.', -2)     AS last_two;    -- 'mysq
```

To get the *middle* piece, nest two calls: `SUBSTRING_INDEX(SUBSTRING_INDEX('a,b,c', ',', 2), ',', -1)` is `'b'`. PostgreSQL has `SPLIT_PART(s, delim, n)` for this, and SQL Server combines `CHARINDEX` with `SUBSTRING`.

Padding and repeating

`LPAD(s, len, fill)` pads on the left until the text is `len` characters long; `RPAD` pads on the right. If the text is already longer, both **cut** it to `len`.

```
SELECT LPAD(42, 6, '0');        -- '000042'
SELECT RPAD('ab', 5, '.');      -- 'ab...'
SELECT LPAD('abcdef', 3, '*'); -- 'abc'  (truncated, not padded)
SELECT REPEAT('-', 5);          -- '-----'
SELECT REVERSE('stressed');     -- 'desserts'
```

Zero-padded codes like `INV-000042` are `CONCAT('INV-', LPAD(id, 6, '0'))`.

Putting them together

Real cleanup chains functions from the inside out. Here is a display name built from untidy input: trim, take the first letter as an initial, and put the surname in capitals.

```
SELECT CONCAT(UPPER(LEFT(TRIM(first_name), 1)), '. ', UPPER(TRIM(last_name)))
FROM people
ORDER BY display;
```

display
A. LOVELACE
G. HOPPER

Read a nested expression from the innermost call outwards: `TRIM(first_name)`, then `LEFT(…, 1)`, then `UPPER(…)`. When a chain grows past three or four levels, give each step a name with a CTE (Module 9) or do the work once when the data is loaded.

Functions in WHERE

String functions work in `WHERE` too: `WHERE LOWER(email) LIKE '%@gmail.com'`, `WHERE CHAR_LENGTH(code) <> 6`. One cost to remember for later: wrapping a column in a function usually stops MySQL from using an index on that column, because the index stores the raw values, not `LOWER(email)`. Module 13 calls such a condition *non-sargable* and shows how to avoid it.

Also note that case sensitivity of `=` depends on the column's **collation**. MySQL's default collations are case-insensitive, so `'ADA' = 'ada'` is true there, while PostgreSQL compares case-sensitively. Normalising with `LOWER` on both sides behaves the same everywhere.

Pitfalls

- `CONCAT` with a NULL argument returns NULL — use `CONCAT_WS` or wrap the column in `COALESCE(col, '')`.
- `||` is OR in MySQL, not concatenation.
- `LENGTH` counts bytes; `CHAR_LENGTH` counts characters.
- Positions are 1-based, and `LOCATE` returns 0 for "not found".
- `REPLACE` is case-sensitive even under a case-insensitive collation.
- `LPAD` / `RPAD` cut text longer than the target length.

Key takeaways

- `CONCAT` joins; `CONCAT_WS` joins with a separator and skips NULLs.

- `TRIM`, `UPPER`, `LOWER` normalise; `CHAR_LENGTH` measures.
- `SUBSTRING`, `LEFT`, `RIGHT` cut by position; `LOCATE` finds; `REPLACE` swaps every match.
- `SUBSTRING_INDEX` splits on a delimiter from either end.
- `LPAD` builds fixed-width codes; functions change the output, never the stored data.

EXERCISE 1

Display names and initials

The `people` table was filled in by hand, so names carry stray spaces and some are missing.

id	first_name	last_name
1	`' Ada '`	`'Lovelace'`
2	`'grace'`	`' Hopper '`
3	`'Alan'`	`'Turing'`

Return `id`, `display` — the trimmed last name, a comma and a space, then the trimmed first name (`Lovelace, Ada`), with a missing part simply left out and no separator for it — and `initials`, the first letter of the trimmed first and last names in capitals (`AL`), again skipping a missing part. Order by `id`.

```
id | display | initials
1 | Lovelace, Ada | AL
2 | Hopper, grace | GH
3 | Turing, Alan | AT
```

INPUT 1

```
CREATE TABLE people (id INT PRIMARY KEY,
first_name VARCHAR(40), last_name
VARCHAR(40));
INSERT INTO people VALUES (1, '  Ada ',
'Lovelace'), (2, 'grace', ' Hopper '),
(3, 'Alan', 'Turing');
```

OUTPUT 1

```
id | display | initials
1 | Lovelace, Ada | AL
2 | Hopper, grace | GH
3 | Turing, Alan | AT
```

REFERENCE SOLUTION

```sql
SELECT id,
       CONCAT_WS(', ', TRIM(last_name), TRIM(first_name))                    AS disp
       UPPER(CONCAT_WS('', LEFT(TRIM(first_name), 1), LEFT(TRIM(last_name), 1))) AS
FROM people
ORDER BY id;
```

EXERCISE 2

Email parts and a masked address

Split each address in `users` into its parts. Addresses may have spaces around them and capital letters; normalise each with `LOWER(TRIM(email))` first.

id	email
1	`' Ada@Example.com'`
2	`'grace@navy.mil'`
3	`'ALAN@example.com '`

Return `id`, `username` (the part before `@`), `domain` (the part after it) and `masked` — the username's first letter, then `***@`, then the domain (`a***@example.com`). Order by `domain`, then `username`.

```
id | username | domain | masked
1 | ada | example.com | a***@example.com
3 | alan | example.com | a***@example.com
2 | grace | navy.mil | g***@navy.mil
```

INPUT 1

```sql
CREATE TABLE users (id INT PRIMARY KEY,
email VARCHAR(80));
INSERT INTO users VALUES (1, '
Ada@Example.com'), (2,
'grace@navy.mil'), (3, 'ALAN@example.com
');
```

OUTPUT 1

```
id | username | domain | masked
1 | ada | example.com | a***@example.com
3 | alan | example.com |
a***@example.com
2 | grace | navy.mil | g***@navy.mil
```

REFERENCE SOLUTION

```sql
SELECT id,
       SUBSTRING_INDEX(LOWER(TRIM(email)), '@', 1)  AS username,
       SUBSTRING_INDEX(LOWER(TRIM(email)), '@', -1) AS domain,
       CONCAT(LEFT(LOWER(TRIM(email)), 1), '***@', SUBSTRING_INDEX(LOWER(TRIM(email)
FROM users
ORDER BY domain, username;
```

QUIZ

1. What does `CONCAT('Ada', NULL, 'Lovelace')` return in MySQL?

A. `'AdaLovelace'`

B. `'Ada Lovelace'`

C. `NULL`

D. An error

2. What does `SUBSTRING_INDEX('www.mysql.com', '.', -2)` return?

A. `'www.mysql'`

B. `'mysql.com'`

C. `'com'`

D. `'mysql'`

3. A UTF-8 column holds `'café'`. What are `LENGTH` and `CHAR_LENGTH` in MySQL?

A. 4 and 4

B. 5 and 4

C. 4 and 5

D. 5 and 5

4. What does `LOCATE('z', 'hello')` return?

A. `-1`

B. `0`

C. `NULL`

D. `6`

5. In MySQL with default settings, what is `'a' || 'b'`?

A. `'ab'`

B. `0` — `||` is logical OR

C. `'a b'`

D. A syntax error

ANSWER KEY · STRING FUNCTIONS — CONCAT, SUBSTRING, TRIM AND FRIENDS

1. C `CONCAT` returns NULL when any argument is NULL. `CONCAT_WS` is the function that skips NULL arguments; nothing here adds a space, and nothing raises an error.

2. B A negative count takes everything after the second delimiter counted from the right: `mysql.com`. A positive 2 would give `www.mysql`; `-1` would give `com`.

3. B `LENGTH` counts bytes and `é` takes two bytes in UTF-8, so 5; `CHAR_LENGTH` counts characters, so 4. For ASCII-only text the two agree.

4. B Positions start at 1, so 0 is never a real position and is what `LOCATE` (and `INSTR`) return for "not found". `-1` is what many programming languages use, not SQL.

5. B In MySQL `||` is OR unless the `PIPES_AS_CONCAT` SQL mode is on, and two non-numeric strings are both false, so the answer is 0. PostgreSQL and standard SQL use `||` for concatenation; in MySQL write `CONCAT`.

LESSON 2 · 13 MIN

Numeric functions — ROUND, CEIL, FLOOR, MOD and division

Numbers rarely leave a database exactly as they are stored. A price gets tax added and needs two decimals; an average needs rounding before anyone reads it; a count of items has to become a count of pages; a sequence number has to be sorted into one of four buckets. This lesson covers MySQL's arithmetic, the four rounding functions and how they differ, division in its decimal and integer forms, remainders, and the handful of other numeric functions that turn up in interview questions.

Arithmetic and the decimal `/`

The operators are the usual ones: `+`, `-`, `*`, `/`, plus `DIV` and `MOD` (also `%`). Arithmetic works on columns and literals alike and follows normal precedence.

```
SELECT name, price, qty, price * qty AS line_total
FROM order_lines
ORDER BY name;
```

The one operator to watch is `/`. In MySQL it is **always decimal division**:

```
SELECT 5 / 2, 6 / 3, 1 / 3;
```

5 / 2	6 / 3	1 / 3
2.5	2	0.3333

MySQL shows `2.5000`, `2.0000` and `0.3333` — four decimal places by default (`div_precision_increment`); this course prints numbers with trailing zeros dropped. In **PostgreSQL, SQL Server and SQLite**, dividing two integers is integer division: `5 / 2` is `2`, and `1 / 3` is `0`. That difference produces the most common wrong answer in percentage questions — a "conversion rate" of 0 for every row. The portable fix is to make one operand decimal before dividing: `100.0 * converted / visits`.

Division by zero in MySQL returns **NULL** (with a warning), not an error. PostgreSQL and SQL Server raise an error, which is why careful queries guard the divisor with `NULLIF(divisor, 0)`

— the conditional-functions lesson shows the idiom.

Integer division: DIV

`DIV` divides and drops the fraction, truncating towards zero:

```
SELECT 17 DIV 5, -17 DIV 5, 17 MOD 5;
```

17 DIV 5	-17 DIV 5	17 MOD 5
3	-3	2

`DIV` and `MOD` go together: `a = b * (a DIV b) + MOD(a, b)`. Seventeen items in boxes of five is three full boxes and two left over.

Remainders: MOD

`MOD(a, b)` returns what is left after dividing. The classic uses:

```
SELECT id FROM tickets WHERE MOD(id, 2) = 0;          -- even ids
SELECT id, MOD(id, 4) + 1 AS team FROM players;       -- round-robin teams 1
SELECT minutes DIV 60 AS h, MOD(minutes, 60) AS m      -- 135 → 2 h 15 m
FROM calls;
```

The result takes the sign of the dividend: `MOD(-7, 3)` is `-1`. `MOD(x, 0)` is NULL in MySQL. "Find the rows with an odd id" (`MOD(id, 2) = 1`) is a common warm-up interview question.

Rounding: ROUND

`ROUND(x, d)` rounds to `d` decimal places; `ROUND(x)` rounds to a whole number. Halves go away from zero for exact decimal values: `ROUND(2.5)` is 3 and `ROUND(-2.5)` is -3.

```
SELECT ROUND(3.14159, 2), ROUND(2.5), ROUND(-2.5), ROUND(19.999, 1);
```

ROUND(3.14159, 2)	ROUND(2.5)	ROUND(-2.5)	ROUND(19.999, 1)
3.14	3	-3	20

A negative `d` rounds to the left of the decimal point in MySQL: `ROUND(1234, -2)` is 1200. Two more facts worth knowing for interviews:

- On `FLOAT` / `DOUBLE` columns, MySQL rounds halves to the nearest even value and binary representation can make `ROUND(1.005, 2)` give `1.00`. On `DECIMAL` columns rounding is exact. This is one of the reasons money is stored as `DECIMAL`.
- Round **at the end**. Rounding each line and then summing gives a different total from summing and then rounding, and the second is what an invoice means.

Up, down and cut: CEIL, FLOOR, TRUNCATE

`CEIL` (or `CEILING`) moves up to the next whole number, `FLOOR` down to the previous one, and `TRUNCATE(x, d)` cuts after `d` decimals without rounding:

```
SELECT CEIL(4.1), FLOOR(4.9), CEIL(-4.1), FLOOR(-4.1), TRUNCATE(4.789, 1), TRU
```

CEIL(4.1)	FLOOR(4.9)	CEIL(-4.1)	FLOOR(-4.1)	TRUNCATE(4.789, 1)	TRUNCATE(-4.789, 1)
5	4	-4	-5	4.7	-4.7

For negative numbers, `FLOOR` goes further from zero and `TRUNCATE` goes towards it — `FLOOR(-4.1)` is -5, `TRUNCATE(-4.1, 0)` is -4. PostgreSQL calls `TRUNCATE` `TRUNC`; SQL Server uses `ROUND(x, d, 1)`.

`CEIL` is the function for "how many containers": 23 results at 10 a page need `CEIL(23 / 10)` = 3 pages. `FLOOR` is the function for buckets: `FLOOR(age / 10) * 10` turns 37 into the 30s.

```
SELECT name, age, FLOOR(age / 10) * 10 AS decade
FROM members
ORDER BY age;
```

name	age	decade
Ravi	19	10
Mia	30	30
Sam	37	30

The rest of the toolbox

```sql
SELECT ABS(-12),          -- 12: distance regardless of sign
       SIGN(-12),         -- -1, 0 or 1
       POWER(2, 10),      -- 1024 (also POW)
       SQRT(144),         -- 12
       GREATEST(3, 9, 4), -- 9: the largest argument
       LEAST(3, 9, 4);    -- 3: the smallest argument
```

`ABS` is the usual way to measure a difference without caring about direction — "orders whose amount differs from the quote by more than 5" is `ABS(amount - quote) > 5`. `GREATEST` and `LEAST` compare across *columns in one row*, unlike `MAX` and `MIN`, which compare one column across *rows* (next module). They return NULL if any argument is NULL. MySQL also has `FORMAT(x, d)`, which returns *text* with thousands separators — `FORMAT(1234567.891, 2)` is `'1,234,567.89'` — handy for display, useless for further arithmetic or sorting.

Types: INT, DECIMAL, FLOAT

The type of a column decides what arithmetic can do with it:

- `INT` and `BIGINT` hold whole numbers exactly.
- `DECIMAL(p, s)` holds exactly `p` digits, `s` of them after the point. `DECIMAL(10, 2)` stores up to 99,999,999.99 exactly. Use it for money.
- `FLOAT` and `DOUBLE` are binary approximations: fast, wide-ranging, and unable to store 0.1 exactly. Use them for measurements, never for amounts that must add up to the cent.

`CAST(x AS DECIMAL(10, 2))` converts and rounds to two places in MySQL; `CAST(x AS SIGNED)` converts to an integer. Strings that look like numbers are converted automatically in arithmetic (`'3' + 4` is 7), which is convenient and also how a typo in a numeric column hides — `'12abc' + 0` is 12, with a warning.

Pitfalls

- Integer division in PostgreSQL, SQL Server and SQLite: multiply by `1.0` first when the answer should have a fraction.
- Rounding too early and summing rounded values.
- `FLOOR` and `TRUNCATE` disagree on negatives.
- Storing money in `FLOAT`.
- Using `FORMAT` and then trying to sort or add the result — it is text.
- `GREATEST` / `LEAST` return NULL when any argument is NULL.

Key takeaways

- MySQL's `/` is decimal; `DIV` is integer division and `MOD` the remainder.
- `ROUND` to the nearest, `CEIL` up, `FLOOR` down, `TRUNCATE` towards zero.
- `CEIL(n / size)` counts containers; `FLOOR(x / w) * w` builds buckets.
- `ABS`, `POWER`, `SQRT`, `GREATEST` and `LEAST` cover most of the rest.
- Money is `DECIMAL`; round once, at the end.

EXERCISE 1

Prices with tax

Each product has a price and a tax rate in percent.

name	price	tax_rate
Desk	250.00	18
Lamp	39.50	12
Notebook	4.20	5
Chair	120.00	18

Return `name`, `price`, `tax` — the tax amount rounded to 2 decimals — and `total` — price plus tax, computed from the unrounded tax and rounded to 2 decimals. Order by `total` descending, then `name`.

```
name | price | tax | total
Desk | 250 | 45 | 295
Chair | 120 | 21.6 | 141.6
Lamp | 39.5 | 4.74 | 44.24
Notebook | 4.2 | 0.21 | 4.41
```

(Numbers print without trailing zeros here; MySQL itself would show `45.00`.)

INPUT 1

```
CREATE TABLE products (name VARCHAR(40),
price DECIMAL(10,2), tax_rate INT);
INSERT INTO products VALUES ('Desk',
250.00, 18), ('Lamp', 39.50, 12),
('Notebook', 4.20, 5), ('Chair', 120.00,
18);
```

OUTPUT 1

```
name | price | tax | total
Desk | 250 | 45 | 295
Chair | 120 | 21.6 | 141.6
Lamp | 39.5 | 4.74 | 44.24
Notebook | 4.2 | 0.21 | 4.41
```

REFERENCE SOLUTION

```sql
SELECT name,
       price,
       ROUND(price * tax_rate / 100, 2)         AS tax,
       ROUND(price + price * tax_rate / 100, 2) AS total
FROM products
ORDER BY total DESC, name;
```

EXERCISE 2

Boxes and leftovers

A warehouse packs `units` items into boxes of `per_box`.

id	units	per_box
1	23	10
2	40	8
3	5	12

Return `id`, `full_boxes` (how many boxes are filled completely), `loose` (items left over) and `boxes_needed` (boxes to ship everything, the last one possibly part-full). Use `DIV`, `MOD` and `CEIL`. Order by `id`. A `per_box` of 0 is a data error — MySQL returns NULL for a division or remainder by zero, and so should your query, without a special case.

```
id | full_boxes | loose | boxes_needed
1 | 2 | 3 | 3
2 | 5 | 0 | 5
3 | 0 | 5 | 1
```

INPUT 1

```
CREATE TABLE shipments (id INT PRIMARY
KEY, units INT, per_box INT);
INSERT INTO shipments VALUES (1, 23,
10), (2, 40, 8), (3, 5, 12);
```

OUTPUT 1

```
id | full_boxes | loose | boxes_needed
1 | 2 | 3 | 3
2 | 5 | 0 | 5
3 | 0 | 5 | 1
```

REFERENCE SOLUTION

```sql
SELECT id,
       units DIV per_box       AS full_boxes,
       MOD(units, per_box)     AS loose,
       CEIL(units / per_box)   AS boxes_needed
FROM shipments
ORDER BY id;
```

QUIZ

1. What does `SELECT 7 / 2;` return in MySQL, and in PostgreSQL?

A. 3.5 in both

B. 3 in both

C. 3.5 in MySQL, 3 in PostgreSQL

D. 3 in MySQL, 3.5 in PostgreSQL

2. What are `FLOOR(-4.2)` and `TRUNCATE(-4.2, 0)`?

A. -4 and -4

B. -5 and -4

C. -5 and -5

D. -4 and -5

3. 47 search results are shown 10 to a page. Which expression gives the number of pages?

A. `47 DIV 10`

B. `ROUND(47 / 10)`

C. `CEIL(47 / 10)`

D. `MOD(47, 10)`

4. Why is money stored as `DECIMAL(10, 2)` rather than `DOUBLE`?

A. `DOUBLE` cannot hold numbers above a million

B. `DECIMAL` stores the digits exactly; `DOUBLE` is a binary approximation that cannot store values like 0.1 exactly

C. `DECIMAL` is faster for arithmetic

D. `DOUBLE` has no negative numbers

ANSWER KEY · NUMERIC FUNCTIONS — ROUND, CEIL, FLOOR, MOD AND DIVISION

1. C MySQL's `/` is always decimal division (it shows `3.5000`). PostgreSQL, SQL Server and SQLite divide two integers as integers and drop the fraction. Multiply by `1.0` first for a decimal answer there.

2. B `FLOOR` always goes down, which for a negative number is away from zero: -5. `TRUNCATE` cuts the digits off, which moves towards zero: -4. For positive numbers the two agree.

3. C Five pages are needed (the fifth holds 7). `CEIL` rounds 4.7 up to 5. `DIV` gives 4 full pages, `ROUND` happens to give 5 here but gives 4 for 41 results, and `MOD` is the 7 on the last page.

4. B Binary floating point stores most decimal fractions approximately, so sums drift by tiny amounts and equality tests fail. `DECIMAL` is exact. `DOUBLE` has a huge range and negatives, and is usually *faster*, which is why it suits measurements.

LESSON 3 · 14 MIN

Dates and times — types, parts and DATE_FORMAT

Almost every business question has a date in it: sales this quarter, users who signed up in March, orders placed on weekends, the busiest hour of the day. MySQL stores dates in dedicated types and offers a function for every part of them, plus one function, `DATE_FORMAT`, that lays a date out as any text you need. This lesson covers the types, reading parts out of a date, formatting, and filtering by date — including the range mistake that quietly drops a day of data. The next lesson does arithmetic with dates.

The types

Type	Holds	Example literal
`DATE`	a calendar day	`'2024-03-15'`
`TIME`	a time of day or a duration	`'14:30:00'`
`DATETIME`	a day and a time, stored as given	`'2024-03-15 14:30:00'`
`TIMESTAMP`	a moment, stored in UTC	`'2024-03-15 14:30:00'`
`YEAR`	a year	`2024`

Date literals are written as strings in `'YYYY-MM-DD'` form, and times as `'HH:MM:SS'`. That ISO order matters: written that way, dates sort and compare correctly even as plain text, which is why every system that stores dates as strings uses it.

`DATETIME` and `TIMESTAMP` look the same in a result but behave differently. A `TIMESTAMP` is converted from the connection's time zone to UTC when stored and back when read, so two users in different zones see the same moment in their own local time; it also only reaches January 2038. A `DATETIME` is stored and returned exactly as written. A rule of thumb: `TIMESTAMP` (or `DATETIME` in UTC by convention) for "when did this happen", `DATE` for "which day is this about" — a birthday or a due date.

The current date and time

```sql
SELECT CURDATE(), NOW(), CURRENT_DATE, CURRENT_TIMESTAMP;
```

`CURDATE()` is today; `NOW()` is the date and time the statement started. Their value depends on the clock, so their output differs every day — this course's exercises never print them, and you will see stored dates compared with fixed literals instead. In real reports they appear in filters: `WHERE created_at ⩾ NOW() - INTERVAL 1 DAY`.

Taking a date apart

Each part has its own function:

```sql
SELECT YEAR('2024-03-15')      AS y,     -- 2024
       MONTH('2024-03-15')     AS m,     -- 3
       DAY('2024-03-15')       AS d,     -- 15
       QUARTER('2024-03-15')   AS q,     -- 1
       DAYNAME('2024-03-15')   AS dn,    -- 'Friday'
       MONTHNAME('2024-03-15') AS mn,    -- 'March'
       DAYOFYEAR('2024-03-15') AS doy;   -- 75
```

For a `DATETIME`, `HOUR`, `MINUTE` and `SECOND` read the time, and `DATE(dt)` drops the time to leave the day: `DATE('2024-03-15 14:30:00')` is `'2024-03-15'`.

Weekdays come in two numberings, and mixing them up is a classic bug:

- `DAYOFWEEK(d)` is 1 for **Sunday** through 7 for Saturday (the ODBC standard).
- `WEEKDAY(d)` is 0 for **Monday** through 6 for Sunday.

So weekends are `DAYOFWEEK(d) IN (1, 7)` or `WEEKDAY(d) ⩾ 5`. When in doubt, compare `DAYNAME(d)` with `'Saturday'` — slower to type, impossible to misread.

The standard SQL spelling is `EXTRACT(part FROM d)` — `EXTRACT(YEAR FROM d)`, `EXTRACT(MONTH FROM d)` — which MySQL and PostgreSQL both accept. SQL Server uses `YEAR(d)`, `MONTH(d)` and `DATEPART(weekday, d)`.

Formatting: DATE_FORMAT

`DATE_FORMAT(d, format)` returns text, laid out by `%` specifiers. The ones you will use most:

Specifier	Meaning	For 2024-03-05 14:07
`%Y`	four-digit year	`2024`
`%y`	two-digit year	`24`
`%m`	month, two digits	`03`
`%c`	month, no padding	`3`
`%M` / `%b`	month name / short	`March` / `Mar`
`%d` / `%e`	day, two digits / no padding	`05` / `5`
`%W` / `%a`	weekday name / short	`Tuesday` / `Tue`
`%H` / `%i` / `%s`	hour (24h), minutes, seconds	`14` / `07` / `00`
`%h` / `%p`	hour (12h), AM or PM	`02` / `PM`

```
SELECT DATE_FORMAT('2024-03-05', '%Y-%m')         AS month_key,   -- '2024-03'
       DATE_FORMAT('2024-03-05', '%d %b %Y')      AS pretty,      -- '05 Mar 20
       DATE_FORMAT('2024-03-05 14:07:00', '%h:%i %p') AS clock;   -- '02:07 PM
```

`DATE_FORMAT(d, '%Y-%m')` is the standard way to turn a date into a *month key* — a value every date in the same month shares, ideal for grouping by month in the next module. Note `%i` is minutes; `%m` is the month, a mistake everyone makes once.

The result of `DATE_FORMAT` is text. Sorting by `'%d %b %Y'` text sorts "01 Apr" before "05 Mar"; sort by the date column itself and only *display* the formatted value. PostgreSQL's equivalent is `TO_CHAR(d, 'YYYY-MM')` and SQL Server's `FORMAT(d, 'yyyy-MM')`. Going the other way, from text to a date, MySQL has `STR_TO_DATE('15/03/2024', '%d/%m/%Y')`.

Filtering by date

Comparison operators work on dates as you would expect:

```
SELECT id, order_date
FROM orders
WHERE order_date ⩾ '2024-03-01' AND order_date < '2024-04-01'
ORDER BY order_date, id;
```

That pattern — **greater than or equal to the first day, less than the first day of the next period** — is the reliable way to ask for a month, a quarter or a year. It is called a half-open range, and it has three advantages:

1. It works for `DATE` and `DATETIME` alike. `BETWEEN '2024-03-01' AND '2024-03-31'` on a `DATETIME` stops at midnight *starting* 31 March and silently drops every order placed later that day.
2. It never needs to know how many days a month has.
3. It compares the column itself, so an index on `order_date` can be used (Module 13).

The alternative, `WHERE YEAR(order_date) = 2024 AND MONTH(order_date) = 3`, gives the same rows and reads naturally, but wraps the column in functions, which rules out the index. For a small table it does not matter; for a large one it is the difference between a lookup and a scan. Function-based filters are the right tool when the question is about a *part* regardless of the period — "all orders ever placed on a Sunday" is `WHERE DAYNAME(order_date) = 'Sunday'`, and there is no range for that.

Dates stored as text

Old schemas sometimes store dates in `VARCHAR` columns. If they are in `'YYYY-MM-DD'` form, comparisons and sorting still work, and MySQL's date functions accept them. If they are in `'15/03/2024'` form, nothing sorts correctly — convert them once with `STR_TO_DATE` into a real `DATE` column. A date type also rejects impossible values like `'2024-02-30'`, which a text column accepts without complaint.

Pitfalls

- `BETWEEN` with a `DATETIME` column drops the last day after midnight.
- `DAYOFWEEK` starts at Sunday = 1; `WEEKDAY` starts at Monday = 0.
- `%m` is the month and `%i` the minutes in `DATE_FORMAT`.
- Sorting by formatted text instead of by the date.
- Printing `NOW()` and expecting the same output tomorrow.
- `TIMESTAMP` values shift with the session time zone; `DATETIME` values do not.

Key takeaways

- Literals are `'YYYY-MM-DD'` and `'YYYY-MM-DD HH:MM:SS'`.
- `YEAR`, `MONTH`, `DAY`, `QUARTER`, `DAYNAME`, `HOUR` read the parts; `DATE()` drops the time.
- `DATE_FORMAT(d, '%Y-%m')` builds month keys; the result is text.
- Filter periods with `≥ start AND < next_start`.
- `CURDATE()` and `NOW()` read the clock and belong in filters, not in judged output.

EXERCISE 1

Month keys and readable dates

Lay out each order's date three ways.

id	order_date
1	2024-03-05
2	2024-01-31
3	2023-12-25
4	2024-03-05

Return `id`, `month_key` (`2024-03`), `pretty` — the short weekday, two-digit day, short month and year (`Tue 05 Mar 2024`) — and `quarter`. Order by `order_date`, then `id`.

```
id | month_key | pretty | quarter
3 | 2023-12 | Mon 25 Dec 2023 | 4
2 | 2024-01 | Wed 31 Jan 2024 | 1
1 | 2024-03 | Tue 05 Mar 2024 | 1
4 | 2024-03 | Tue 05 Mar 2024 | 1
```

INPUT 1

```
CREATE TABLE orders (id INT PRIMARY KEY,
order_date DATE);
INSERT INTO orders VALUES (1, '2024-03-
05'), (2, '2024-01-31'), (3, '2023-12-
25'), (4, '2024-03-05');
```

OUTPUT 1

```
id | month_key | pretty | quarter
3 | 2023-12 | Mon 25 Dec 2023 | 4
2 | 2024-01 | Wed 31 Jan 2024 | 1
1 | 2024-03 | Tue 05 Mar 2024 | 1
4 | 2024-03 | Tue 05 Mar 2024 | 1
```

REFERENCE SOLUTION

```sql
SELECT id,
       DATE_FORMAT(order_date, '%Y-%m')          AS month_key,
       DATE_FORMAT(order_date, '%a %d %b %Y')    AS pretty,
       QUARTER(order_date)                       AS quarter
FROM orders
ORDER BY order_date, id;
```

EXERCISE 2

Weekend orders in March

`placed_at` is a `DATETIME`. Find the orders placed in **March 2024** on a **Saturday or Sunday**.

id	placed_at	amount
1	2024-03-02 10:15:00	40
2	2024-03-04 09:00:00	25
3	2024-03-31 18:30:00	60
4	2024-02-25 12:00:00	15
5	2024-04-06 11:00:00	30
6	2024-03-16 08:00:00	20

Return `id`, `placed_at` and `day_name` (`Saturday`). Select the month with a half-open range on `placed_at` — an order late on 31 March must be included. Order by `placed_at`, then `id`.

```
id | placed_at | day_name
1 | 2024-03-02 10:15:00 | Saturday
6 | 2024-03-16 08:00:00 | Saturday
3 | 2024-03-31 18:30:00 | Sunday
```

INPUT 1

```
CREATE TABLE orders (id INT PRIMARY KEY,
placed_at DATETIME, amount INT);
INSERT INTO orders VALUES (1, '2024-03-
02 10:15:00', 40), (2, '2024-03-04
09:00:00', 25), (3, '2024-03-31
18:30:00', 60),
  (4, '2024-02-25 12:00:00', 15), (5,
'2024-04-06 11:00:00', 30), (6, '2024-
03-16 08:00:00', 20);
```

OUTPUT 1

```
id | placed_at | day_name
1 | 2024-03-02 10:15:00 | Saturday
6 | 2024-03-16 08:00:00 | Saturday
3 | 2024-03-31 18:30:00 | Sunday
```

REFERENCE SOLUTION

```sql
SELECT id, placed_at, DAYNAME(placed_at) AS day_name
FROM orders
WHERE placed_at ⩾ '2024-03-01' AND placed_at < '2024-04-01'
  AND DAYOFWEEK(placed_at) IN (1, 7)
ORDER BY placed_at, id;
```

QUIZ

1. Which condition selects every row of March 2024 from a `DATETIME` column `created_at`, and can use an index on it?

A. `created_at BETWEEN '2024-03-01' AND '2024-03-31'`

B. `created_at ⩾ '2024-03-01' AND created_at < '2024-04-01'`

C. `MONTH(created_at) = 3`

D. `DATE_FORMAT(created_at, '%Y-%m') = '2024-03'`

2. What does `DAYOFWEEK('2024-03-03')` return? (3 March 2024 was a Sunday.)

A. 0

B. 1

C. 6

D. 7

3. What does `DATE_FORMAT('2024-03-05 14:07:00', '%Y-%m-%d %H:%m')` return?

A. `'2024-03-05 14:07'`

B. `'2024-03-05 14:03'`

C. `'2024-03-05 02:07'`

D. An error

4. Which type converts its value to UTC on storage and back to the session's time zone on reading?

A. `DATE`

B. `DATETIME`

C. `TIMESTAMP`

D. `TIME`

ANSWER KEY · DATES AND TIMES — TYPES, PARTS AND DATE_FORMAT

1. B The half-open range includes all of 31 March and compares the bare column, so an index applies. `BETWEEN` stops at midnight starting the 31st; `MONTH(…) = 3` matches March of every year; the `DATE_FORMAT` version is correct but wraps the column in a function, which rules out the index.

2. B `DAYOFWEEK` numbers Sunday as 1 through Saturday as 7. `WEEKDAY` uses a different numbering — Monday 0 through Sunday 6 — and would return 6.

3. B `%m` is the month in `DATE_FORMAT`, so the time shows `14:03`. Minutes are `%i`. A classic slip; the function does not raise an error for it.

4. C `TIMESTAMP` stores a moment in UTC and converts using the connection's time zone; it also only reaches 2038. `DATETIME` stores exactly what was written, and `DATE` / `TIME` hold a day or a time of day.

LESSON 4 · 14 MIN

Date arithmetic — INTERVAL, DATEDIFF and TIMESTAMPDIFF

Reading parts out of a date answers "when". Arithmetic answers "how long" and "by when": the due date of a loan, the number of days an invoice is overdue, a subscription's renewal, a customer's tenure in months, the users who came back within seven days. MySQL gives you three tools — `INTERVAL` to move a date, `DATEDIFF` to count days, and `TIMESTAMPDIFF` to count any unit — and one helper, `LAST_DAY`, for the month edges that make calendar arithmetic awkward.

Moving a date: DATE_ADD and INTERVAL

`DATE_ADD(d, INTERVAL n unit)` returns the date moved forward by `n` units; `DATE_SUB` moves it back. The units are `SECOND`, `MINUTE`, `HOUR`, `DAY`, `WEEK`, `MONTH`, `QUARTER` and `YEAR`.

```
SELECT DATE_ADD('2024-03-15', INTERVAL 10 DAY)    AS plus_10_days,   -- '2024
       DATE_SUB('2024-03-15', INTERVAL 2 WEEK)    AS minus_2_weeks,  -- '2024
       DATE_ADD('2024-03-15', INTERVAL 1 YEAR)    AS next_year,      -- '2025
       DATE_ADD('2024-03-15 23:30:00', INTERVAL 45 MINUTE) AS later; -- '2024
```

MySQL also accepts the operator form: `order_date + INTERVAL 30 DAY` and `order_date - INTERVAL 1 MONTH` mean the same thing and read well inside a `WHERE`. A negative amount moves the other way, so `DATE_ADD(d, INTERVAL -7 DAY)` is a week earlier.

Adding to a `DATE` gives a `DATE`; adding hours or minutes, or adding to a `DATETIME`, gives a `DATETIME`. Never add a bare number to a date: `'2024-03-15' + 1` is not the 16th — MySQL turns the string into the number 2024 and adds 1. Always say `INTERVAL`.

Months are not all the same length

Adding days is simple arithmetic. Adding months is calendar arithmetic, because a month has 28 to 31 days. MySQL's rule: keep the day number, and if the target month is too short, **clamp to its last day**.

```sql
SELECT DATE_ADD('2024-01-31', INTERVAL 1 MONTH) AS a,   -- '2024-02-29' (leap
       DATE_ADD('2023-01-31', INTERVAL 1 MONTH) AS b,   -- '2023-02-28'
       DATE_ADD('2024-02-29', INTERVAL 1 YEAR)  AS c,   -- '2025-02-28'
       DATE_ADD('2024-03-31', INTERVAL -1 MONTH) AS d;  -- '2024-02-29'
```

A consequence worth remembering: adding one month twice is not always adding two months. `'2024-01-31'` plus one month is `'2024-02-29'`, plus another is `'2024-03-29'` — but plus two months at once is `'2024-03-31'`. Billing systems compute every renewal from the original start date for this reason, not from the previous renewal.

Counting days: DATEDIFF

`DATEDIFF(a, b)` returns `a - b` in days. The later date goes **first** for a positive answer, and any time part is ignored.

```sql
SELECT DATEDIFF('2024-03-15', '2024-03-01') AS a,          -- 14
       DATEDIFF('2024-03-01', '2024-03-15') AS b,          -- -14
       DATEDIFF('2024-03-01', '2024-02-01') AS c,          -- 29 (leap Februa
       DATEDIFF('2024-03-02 00:01:00', '2024-03-01 23:59:00') AS d;   -- 1
```

The last line shows what "ignores the time" means: two minutes apart, but on different days, so the answer is 1. `DATEDIFF` counts calendar days, which is usually what a business question means by "days late".

A typical use is an overdue report:

```sql
SELECT id, due_date, returned_on,
       DATEDIFF(returned_on, due_date) AS days_late
FROM loans
WHERE returned_on > due_date
ORDER BY days_late DESC, id;
```

Counting any unit: TIMESTAMPDIFF

`TIMESTAMPDIFF(unit, start, end)` returns `end - start` in the given unit, counting **whole** units only. Note the argument order is the opposite of `DATEDIFF`: start first, end second.

```sql
SELECT TIMESTAMPDIFF(MINUTE, '2024-03-15 09:00:00', '2024-03-15 10:45:00') AS
       TIMESTAMPDIFF(HOUR,   '2024-03-15 09:00:00', '2024-03-15 10:45:00') AS
       TIMESTAMPDIFF(MONTH,  '2024-01-15', '2024-03-14') AS months_a,
       TIMESTAMPDIFF(MONTH,  '2024-01-15', '2024-03-15') AS months_b,
       TIMESTAMPDIFF(YEAR,   '2000-06-20', '2024-06-19') AS age;
```

"Whole units" is exactly the rule for ages and tenure: someone born on 20 June 2000 is still 23 on 19 June 2024. Dividing `DATEDIFF` by 365 gets birthdays wrong and drifts with leap years; `TIMESTAMPDIFF(YEAR, …)` does not.

Month edges: LAST_DAY and the first of the month

`LAST_DAY(d)` returns the last day of `d`'s month, leap years included. The first day of a month has no dedicated MySQL function, but two idioms give it:

```sql
SELECT LAST_DAY('2024-02-10')                          AS month_end,     -- '202
       DATE_FORMAT('2024-02-10', '%Y-%m-01')           AS month_start,   -- '202
       DATE_SUB('2024-02-10', INTERVAL DAY('2024-02-10') - 1 DAY) AS also_star
       DATE_ADD(LAST_DAY('2024-02-10'), INTERVAL 1 DAY) AS next_month; -- '202
```

`DAY(LAST_DAY(d))` is the number of days in the month — a common building block for "average per day this month" and for checking leap years.

Rolling windows

Arithmetic with `INTERVAL` is how relative periods are written. Against a fixed reporting date:

```sql
-- orders in the 7 days up to and including 2024-03-15
SELECT id, order_date
FROM orders
WHERE order_date >  DATE_SUB('2024-03-15', INTERVAL 7 DAY)
  AND order_date ⩽ '2024-03-15'
ORDER BY order_date, id;
```

In production the fixed date is usually `CURDATE()`. Notice the filter keeps the column bare and moves the arithmetic to the constant side — `order_date > CURDATE() - INTERVAL 7 DAY` can use an index, while `DATEDIFF(CURDATE(), order_date) < 7` cannot.

Other databases

Interviewers often ask how this looks elsewhere:

- **PostgreSQL**: `d + INTERVAL '10 days'`, `d2 - d1` gives an integer number of days for `DATE` values, and `AGE(end, start)` returns an interval of years, months and days.
- **SQL Server**: `DATEADD(day, 10, d)` and `DATEDIFF(day, start, end)` — the unit first, then start, then end. It counts *boundaries crossed*, so `DATEDIFF(year, '2023-12-31', '2024-01-01')` is 1, unlike MySQL's `TIMESTAMPDIFF(YEAR, …)`, which is 0.
- **Standard SQL** has `INTERVAL` literals too; the MySQL `INTERVAL n UNIT` form is close to it.

Pitfalls

- `DATEDIFF(later, earlier)` but `TIMESTAMPDIFF(unit, earlier, later)` — opposite orders.
- Adding a bare number to a date instead of an `INTERVAL`.
- Expecting `'2024-01-31' + INTERVAL 1 MONTH` to be in March.
- Ages from `DATEDIFF / 365`.
- Putting the column inside the function in a filter when the arithmetic could move to the constant.

Key takeaways

- `DATE_ADD(d, INTERVAL n unit)`, `DATE_SUB`, or `d ± INTERVAL n unit` move a date.
- Month arithmetic clamps to the end of a shorter month.
- `DATEDIFF(a, b)` is calendar days `a - b`; `TIMESTAMPDIFF(unit, start, end)` counts whole units.
- `LAST_DAY(d)` is the month's end; `DATE_FORMAT(d, '%Y-%m-01')` its start.
- Keep the column bare in filters and do the arithmetic on the constant.

EXERCISE 1

Library loans: due dates and days late

Books are lent for 14 days. `returned_on` is NULL while a book is still out.

id	borrowed_on	returned_on
1	2024-03-01	2024-03-10
2	2024-03-01	2024-03-20
3	2024-02-20	NULL
4	2024-02-25	2024-03-15

Return `id`, `due_on` (borrowed date plus 14 days) and `days_late` — days between the due date and the return, never below 0 (an early return is 0 days late), and NULL for a book not yet returned. Order by `id`.

```
id | due_on | days_late
1 | 2024-03-15 | 0
2 | 2024-03-15 | 5
3 | 2024-03-05 | NULL
4 | 2024-03-10 | 5
```

INPUT 1

```
CREATE TABLE loans (id INT PRIMARY KEY,
borrowed_on DATE, returned_on DATE);
INSERT INTO loans VALUES (1, '2024-03-
01', '2024-03-10'), (2, '2024-03-01',
'2024-03-20'), (3, '2024-02-20', NULL),
(4, '2024-02-25', '2024-03-15');
```

OUTPUT 1

```
id | due_on | days_late
1 | 2024-03-15 | 0
2 | 2024-03-15 | 5
3 | 2024-03-05 | NULL
4 | 2024-03-10 | 5
```

REFERENCE SOLUTION

```sql
SELECT id,
       DATE_ADD(borrowed_on, INTERVAL 14 DAY)                                  AS due_
       GREATEST(DATEDIFF(returned_on, DATE_ADD(borrowed_on, INTERVAL 14 DAY)), 0) AS
FROM loans
ORDER BY id;
```

EXERCISE 2

Subscriptions: renewal, tenure and month end

Each subscription has a start and an end date.

id	started_on	ended_on
1	2024-01-31	2024-07-15
2	2023-11-15	2024-02-14
3	2024-02-10	2024-02-29

Return `id`, `renewal` (the start date plus one month, MySQL-style — a 31st clamps to the end of a shorter month), `months_active` (whole months from start to end) and `month_end` (the last day of the starting month). Order by `months_active` descending, then `id`.

```
id | renewal | months_active | month_end
1 | 2024-02-29 | 5 | 2024-01-31
2 | 2023-12-15 | 2 | 2023-11-30
3 | 2024-03-10 | 0 | 2024-02-29
```

INPUT 1

```
CREATE TABLE subscriptions (id INT
PRIMARY KEY, started_on DATE, ended_on
DATE);
INSERT INTO subscriptions VALUES (1,
'2024-01-31', '2024-07-15'), (2, '2023-
11-15', '2024-02-14'), (3, '2024-02-10',
'2024-02-29');
```

OUTPUT 1

```
id | renewal | months_active | month_end
1 | 2024-02-29 | 5 | 2024-01-31
2 | 2023-12-15 | 2 | 2023-11-30
3 | 2024-03-10 | 0 | 2024-02-29
```

REFERENCE SOLUTION

```sql
SELECT id,
       DATE_ADD(started_on, INTERVAL 1 MONTH)         AS renewal,
       TIMESTAMPDIFF(MONTH, started_on, ended_on)     AS months_active,
       LAST_DAY(started_on)                           AS month_end
FROM subscriptions
ORDER BY months_active DESC, id;
```

QUIZ

1. What does `DATE_ADD('2023-01-31', INTERVAL 1 MONTH)` return in MySQL?

A. `'2023-03-03'`

B. `'2023-02-28'`

C. `'2023-03-01'`

D. NULL — 31 February does not exist

2. What does `DATEDIFF('2024-03-01', '2024-03-15')` return?

A. `14`

B. `-14`

C. `0`

D. `NULL`

3. Someone was born on 2000-06-20. Which expression gives their age on 2024-06-19 correctly as 23?

A. `DATEDIFF('2024-06-19', '2000-06-20') / 365`

B. `YEAR('2024-06-19') - YEAR('2000-06-20')`

C. `TIMESTAMPDIFF(YEAR, '2000-06-20', '2024-06-19')`

D. `TIMESTAMPDIFF(YEAR, '2024-06-19', '2000-06-20')`

4. Which filter finds orders from the last 30 days and can still use an index on `order_date`?

A. `WHERE DATEDIFF(CURDATE(), order_date) ⩽ 30`

B. `WHERE order_date ⩾ DATE_SUB(CURDATE(), INTERVAL 30 DAY)`

C. `WHERE TIMESTAMPDIFF(DAY, order_date, CURDATE()) ⩽ 30`

D. All three equally

ANSWER KEY · DATE ARITHMETIC — INTERVAL, DATEDIFF AND TIMESTAMPDIFF

1. B MySQL keeps the day number and clamps it to the last day of a shorter month, so the answer is 28 February (2023 is not a leap year). It never rolls over into March and never returns NULL for this.

2. B `DATEDIFF(a, b)` is `a - b` in days. The first date is earlier, so the result is negative. Put the later date first for a positive count.

3. C `TIMESTAMPDIFF(YEAR, start, end)` counts completed years, so 23. Subtracting years gives 24 before the birthday; dividing days by 365 drifts with leap years and is not a whole number; swapping the dates gives -23.

4. B Moving the arithmetic to the constant side leaves the column bare, so MySQL can range-scan an index. The other two return the same rows but wrap the column in a function, which forces every row to be computed.

LESSON 5 · 13 MIN

Conditional functions — IF, IFNULL, NULLIF, GREATEST and FIELD

The previous module introduced `CASE` and `COALESCE`, the two standard ways to make a decision inside an expression. MySQL adds a set of shorter functions for the decisions that come up constantly: pick one of two values, replace a NULL, turn a sentinel into NULL, take the larger of two columns, sort by a list. They are small, but they make up a large share of real report queries and of interview answers — "avoid division by zero" and "sort status in a custom order" are both one-function answers.

IF: a two-way choice

`IF(cond, a, b)` returns `a` when `cond` is true and `b` otherwise.

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

name	stock	availability
Desk	0	sold out
Lamp	12	in stock

"Otherwise" includes NULL: `IF(NULL > 0, 'yes', 'no')` returns `'no'`, because the condition is not true. That matches `CASE`, which also falls to `ELSE` when no `WHEN` is true.

`IF` is exactly a two-branch `CASE`:

```
IF(stock > 0, 'in stock', 'sold out')
CASE WHEN stock > 0 THEN 'in stock' ELSE 'sold out' END
```

`IF` is MySQL-only (SQL Server has a similar `IIF`); `CASE` is standard. Nesting `IF`s for three or more outcomes — `IF(a, x, IF(b, y, z))` — works but reads badly; once there is a third branch, switch to `CASE`. Do not confuse the `IF()` function with the `IF … THEN … END IF` statement of stored procedures, which is a different thing.

IFNULL and COALESCE: replacing NULL

`IFNULL(a, b)` returns `a` unless it is NULL, in which case `b`. `COALESCE` does the same for any number of arguments and returns the first non-NULL one.

```
SELECT name,
       IFNULL(phone, 'no phone')                AS phone,
       COALESCE(mobile, phone, email, 'none')   AS best_contact
FROM contacts
ORDER BY name;
```

They matter most in arithmetic, where one NULL poisons the whole expression: `price - discount` is NULL for every product with no discount, and `price - IFNULL(discount, 0)` is the price. For portability prefer `COALESCE`; SQL Server's two-argument version is spelled `ISNULL(a, b)` — which in MySQL is a different, one-argument function that returns 1 or 0.

NULLIF: turning a value into NULL

`NULLIF(a, b)` returns NULL when `a = b`, and `a` otherwise. It sounds backwards until you see its two jobs.

Safe division. Dividing by `NULLIF(d, 0)` divides by NULL instead of zero, which gives NULL rather than an error:

```
SELECT campaign,
       ROUND(100 * clicks / NULLIF(views, 0), 1) AS ctr
FROM campaigns
ORDER BY campaign;
```

MySQL returns NULL for a division by zero anyway, but PostgreSQL and SQL Server raise an error and stop the query, so the `NULLIF` guard is the habit worth having everywhere. Wrap the whole expression in `COALESCE(…, 0)` if a zero is wanted instead of NULL.

Cleaning sentinels. Old data often stores "unknown" as an empty string, 0 or `'N/A'`. `NULLIF(col, '')` turns those into real NULLs, so `COALESCE`, `COUNT(col)` and `AVG(col)` treat them as missing:

```
SELECT COALESCE(NULLIF(TRIM(nickname), ''), name) AS shown_name
FROM users;
```

GREATEST and LEAST: across columns

`GREATEST(a, b, …)` returns the largest of its arguments and `LEAST` the smallest. They work *within one row*, across columns or expressions, which is what separates them from the aggregates `MAX` and `MIN` (next module), which work down a column across rows.

```sql
SELECT player,
       GREATEST(round1, round2, round3) AS best_round,
       LEAST(round1, round2, round3)    AS worst_round,
       GREATEST(price - discount, 0)    AS charged      -- never below zero
FROM scores
ORDER BY player;
```

`GREATEST(x, 0)` is the standard way to **clamp** a value at a floor, and `LEAST(x, 100)` at a ceiling. Combine them for a range: `LEAST(GREATEST(score, 0), 100)`.

The NULL rule differs between databases. In MySQL, if **any** argument is NULL the result is NULL; PostgreSQL ignores NULL arguments instead. On MySQL, wrap nullable columns: `GREATEST(COALESCE(a, 0), COALESCE(b, 0))`.

FIELD: position in a list

`FIELD(x, v1, v2, v3, …)` returns the position of `x` in the list — 1 for `v1`, 2 for `v2` — and **0** when `x` is not there. Its main use is sorting in an order that is neither alphabetical nor numeric:

```sql
SELECT id, status
FROM tickets
ORDER BY FIELD(status, 'urgent', 'open', 'waiting', 'closed'), id;
```

Statuses outside the list get 0 and come **first**. To put unknown values last, sort by `FIELD(…) = 0` first — false (0) sorts before true (1) — or list them explicitly. The portable spelling is a `CASE` that maps each value to a number:

```sql
ORDER BY CASE status WHEN 'urgent' THEN 1 WHEN 'open' THEN 2
                    WHEN 'waiting' THEN 3 ELSE 4 END, id
```

Choosing between them

Need	MySQL shorthand	Portable form
two outcomes	`IF(c, a, b)`	`CASE WHEN c THEN a ELSE b END`
three or more outcomes	—	`CASE`
replace NULL	`IFNULL(a, b)`	`COALESCE(a, b)`
make a value NULL	`NULLIF(a, b)`	`NULLIF(a, b)` (standard)
largest/smallest in a row	`GREATEST`, `LEAST`	same in PostgreSQL and Oracle; SQL Server 2022+
custom sort	`FIELD(x, …)`	`CASE x WHEN … THEN 1 … END`

Use the shorthand when it reads better and the code will only run on MySQL; use `CASE` and `COALESCE` when portability or an interviewer's preference for standard SQL matters.

Pitfalls

- `IF` treats a NULL condition as false — make sure that is what you mean.
- Nesting `IF` three deep instead of writing a `CASE`.
- `ISNULL(a, b)` is SQL Server; MySQL's `ISNULL(a)` takes one argument and returns 1 or 0.
- `GREATEST` / `LEAST` return NULL in MySQL if any argument is NULL.
- `FIELD` returns 0 for unlisted values, which sorts them first.
- Dividing by a column that can be zero without `NULLIF`.

Key takeaways

- `IF(c, a, b)` is a two-branch `CASE`; use `CASE` for anything more.
- `IFNULL` / `COALESCE` replace NULLs; `NULLIF` creates them — safe division and sentinel cleanup.
- `GREATEST` / `LEAST` work across columns and clamp values.
- `FIELD` gives a custom sort order; a `CASE` mapping does the same portably.

EXERCISE 1

A stock board in aisle order

Build a stock board for the shop floor.

name	category	stock	reorder_level
Milk	dairy	0	10
Bread	bakery	5	NULL
Apples	fresh	30	20
Cheese	dairy	4	6
Bagels	bakery	0	8

Return `name`, `category`, `availability` — `in stock` when `stock > 0`, otherwise `sold out` — and `shortfall`, how many units are needed to reach the reorder level, never below 0 (a missing reorder level counts as 0). Order the rows by aisle — `fresh`, then `dairy`, then `bakery` — and by `name` within an aisle.

```
name | category | availability | shortfall
Apples | fresh | in stock | 0
Cheese | dairy | in stock | 2
Milk | dairy | sold out | 10
Bagels | bakery | sold out | 8
Bread | bakery | in stock | 0
```

INPUT 1

```
CREATE TABLE products (name VARCHAR(40),
category VARCHAR(20), stock INT,
reorder_level INT);
INSERT INTO products VALUES ('Milk',
'dairy', 0, 10), ('Bread', 'bakery', 5,
NULL), ('Apples', 'fresh', 30, 20),
('Cheese', 'dairy', 4, 6), ('Bagels',
'bakery', 0, 8);
```

OUTPUT 1

```
name | category | availability |
shortfall
Apples | fresh | in stock | 0
Cheese | dairy | in stock | 2
Milk | dairy | sold out | 10
Bagels | bakery | sold out | 8
Bread | bakery | in stock | 0
```

REFERENCE SOLUTION

```sql
SELECT name,
       category,
       IF(stock > 0, 'in stock', 'sold out')          AS availability,
       GREATEST(IFNULL(reorder_level, 0) - stock, 0)  AS shortfall
FROM products
ORDER BY FIELD(category, 'fresh', 'dairy', 'bakery'), name;
```

EXERCISE 2

Click-through rates that survive zero views

Compute each campaign's click-through rate as a percentage.

name	views	clicks
spring	450	37
launch	0	0
retarget	1200	96
email	800	20

Return `name`, `ctr` — `100 * clicks / views` rounded to 1 decimal, NULL when there are no views (guard the divisor with `NULLIF`) — and `ctr_or_zero`, the same value with NULL shown as 0. Order by `ctr_or_zero` descending, then `name`.

```
name | ctr | ctr_or_zero
spring | 8.2 | 8.2
retarget | 8 | 8
email | 2.5 | 2.5
launch | NULL | 0
```

INPUT 1

```
CREATE TABLE campaigns (name
VARCHAR(40), views INT, clicks INT);
INSERT INTO campaigns VALUES ('spring',
450, 37), ('launch', 0, 0), ('retarget',
1200, 96), ('email', 800, 20);
```

OUTPUT 1

```
name | ctr | ctr_or_zero
spring | 8.2 | 8.2
retarget | 8 | 8
email | 2.5 | 2.5
launch | NULL | 0
```

REFERENCE SOLUTION

```sql
SELECT name,
       ROUND(100 * clicks / NULLIF(views, 0), 1)              AS ctr,
       COALESCE(ROUND(100 * clicks / NULLIF(views, 0), 1), 0) AS ctr_or_zero
FROM campaigns
ORDER BY ctr_or_zero DESC, name;
```

QUIZ

1. What does `IF(NULL > 0, 'yes', 'no')` return?

A. `'yes'`

B. `'no'`

C. `NULL`

D. An error

2. What does `NULLIF(5, 5)` return, and `NULLIF(5, 3)`?

A. `0` and `5`

B. `NULL` and `5`

C. `5` and `NULL`

D. `NULL` and `3`

3. In MySQL, what is `GREATEST(4, NULL, 9)`?

A. `9`

B. `4`

C. `NULL`

D. An error

4. A query ends `ORDER BY FIELD(status, 'open', 'closed')`. Where do rows with status `'pending'` appear?

A. Last

B. First

C. Between open and closed

D. They are left out

5. Which is the portable, standard-SQL way to replace a NULL `discount` with 0?

A. `IFNULL(discount, 0)`

B. `ISNULL(discount, 0)`

C. `COALESCE(discount, 0)`

D. `NVL(discount, 0)`

ANSWER KEY · CONDITIONAL FUNCTIONS — IF, IFNULL, NULLIF, GREATEST AND FIELD

1. B `NULL > 0` is unknown, which is not true, so `IF` returns its third argument. `CASE` behaves the same way, falling through to `ELSE`.

2. B `NULLIF(a, b)` returns NULL when the two are equal and `a` otherwise. That is what makes `x / NULLIF(y, 0)` safe.

3. C MySQL's `GREATEST` and `LEAST` return NULL if any argument is NULL. PostgreSQL ignores the NULL and returns 9 — a real difference between the two. Wrap nullable columns in `COALESCE` on MySQL.

4. B `FIELD` returns 0 for a value not in its list, and 0 sorts before 1 and 2, so unlisted values come first. `ORDER BY` never removes rows.

5. C `COALESCE` is standard and works in MySQL, PostgreSQL, SQL Server and Oracle. `IFNULL` is MySQL/SQLite, `ISNULL(a, b)` is SQL Server (MySQL's `ISNULL` takes one argument), and `NVL` is Oracle.

CHECKPOINT · 25 MIN

Checkpoint — Functions

This checkpoint covers the whole module: string functions for joining, cleaning, cutting and padding text; numeric functions for rounding, integer division and remainders; date types, date parts and `DATE_FORMAT`; date arithmetic with `INTERVAL`, `DATEDIFF`, `TIMESTAMPDIFF` and `LAST_DAY`; and the conditional functions `IF`, `IFNULL`, `NULLIF`, `GREATEST`, `LEAST` and `FIELD`.

How it works. Eleven 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 does `CONCAT` return when one argument is NULL, and what does `CONCAT_WS` do instead?
- What do `SUBSTRING_INDEX(s, '@', 1)` and `SUBSTRING_INDEX(s, '@', -1)` return for an email address?
- What is `-7 DIV 2`, and how do `FLOOR`, `CEIL`, `ROUND` and `TRUNCATE` differ for negative numbers?
- Why is `5 / 2` equal to 2 in PostgreSQL and 2.5 in MySQL?
- Which weekday is 1 in `DAYOFWEEK`, and which is 0 in `WEEKDAY`?
- What is `'2024-01-31' + INTERVAL 1 MONTH`?
- What is the argument order of `DATEDIFF`, and of `TIMESTAMPDIFF`?
- Why is a half-open range better than `BETWEEN` for a month of `DATETIME` values?
- What do `NULLIF(x, 0)` and `FIELD(status, …)` return, and what are they used for?

The three queries are a contact-card formatter that cleans names and splits emails, an invoice report that computes due dates, days overdue and a rounded late fee, and a shipment board with labels, a safe on-time rate and a custom sort order.

EXERCISE 1

Contact cards

`contacts.full_name` was typed in any case with stray spaces; the first word is the first name and the last word the surname.

id	full_name	email
1	`' ada LOVELACE '`	Ada@Example.COM
2	`'grace hopper'`	grace@NAVY.mil
3	`'Linus'`	linus@kernel.org

Return `id`, `first_name` and `last_name` — each with a capital first letter and the rest lower case — and `domain`, the lower-cased part of the email after `@`. A one-word name is both the first and the last name. Order by `last_name`, then `id`.

```
id | first_name | last_name | domain
2 | Grace | Hopper | navy.mil
3 | Linus | Linus | kernel.org
1 | Ada | Lovelace | example.com
```

INPUT 1

```
CREATE TABLE contacts (id INT PRIMARY
KEY, full_name VARCHAR(60), email
VARCHAR(80));
INSERT INTO contacts VALUES (1, '  ada
LOVELACE ', 'Ada@Example.COM'), (2,
'grace hopper', 'grace@NAVY.mil'), (3,
'Linus', 'linus@kernel.org');
```

OUTPUT 1

```
id | first_name | last_name | domain
2 | Grace | Hopper | navy.mil
3 | Linus | Linus | kernel.org
1 | Ada | Lovelace | example.com
```

REFERENCE SOLUTION

```
SELECT id,
       CONCAT(UPPER(LEFT(SUBSTRING_INDEX(TRIM(full_name), ' ', 1), 1)),
              LOWER(SUBSTRING(SUBSTRING_INDEX(TRIM(full_name), ' ', 1), 2)))   AS fi
       CONCAT(UPPER(LEFT(SUBSTRING_INDEX(TRIM(full_name), ' ', -1), 1)),
              LOWER(SUBSTRING(SUBSTRING_INDEX(TRIM(full_name), ' ', -1), 2)))  AS la
       LOWER(SUBSTRING_INDEX(email, '@', -1))                                   AS d
FROM contacts
ORDER BY last_name, id;
```

EXERCISE 2

Invoice ageing with late fees

Report on invoices as of **2024-06-30**. An invoice is due `terms_days` after it was issued; an unpaid one (`paid_on` NULL) is measured up to 2024-06-30.

id	issued_on	terms_days	paid_on	amount
1	2024-05-01	30	NULL	1200.00
2	2024-06-01	15	2024-06-10	500.00
3	2024-04-15	45	2024-06-05	250.00
4	2024-06-20	7	NULL	80.00

Return `id` , `due_on` , `days_overdue` — days from the due date to the payment date (or to 2024-06-30), never below 0 — and `late_fee` , 1% of the amount for every week or part-week overdue, rounded to 2 decimals. Order by `days_overdue` descending, then `id` .

```
id | due_on | days_overdue | late_fee
1 | 2024-05-31 | 30 | 60
3 | 2024-05-30 | 6 | 2.5
4 | 2024-06-27 | 3 | 0.8
2 | 2024-06-16 | 0 | 0
```

INPUT 1

```
CREATE TABLE invoices (id INT PRIMARY
KEY, issued_on DATE, terms_days INT,
paid_on DATE, amount DECIMAL(10,2));
INSERT INTO invoices VALUES (1, '2024-
05-01', 30, NULL, 1200.00), (2, '2024-
06-01', 15, '2024-06-10', 500.00),
  (3, '2024-04-15', 45, '2024-06-05',
250.00), (4, '2024-06-20', 7, NULL,
80.00);
```

OUTPUT 1

```
id | due_on | days_overdue | late_fee
1 | 2024-05-31 | 30 | 60
3 | 2024-05-30 | 6 | 2.5
4 | 2024-06-27 | 3 | 0.8
2 | 2024-06-16 | 0 | 0
```

REFERENCE SOLUTION

```
SELECT id,
       DATE_ADD(issued_on, INTERVAL terms_days DAY) AS due_on,
       GREATEST(DATEDIFF(COALESCE(paid_on, '2024-06-30'), DATE_ADD(issued_on, INTERV
       ROUND(amount * 0.01 *
             CEIL(GREATEST(DATEDIFF(COALESCE(paid_on, '2024-06-30'), DATE_ADD(issuec
FROM invoices
ORDER BY days_overdue DESC, id;
```

EXERCISE 3

Carrier scorecard

Grade each delivery carrier on its on-time rate.

name	tier	delivered	on_time
BlueDart	express	200	192
Delhivery	standard	500	410
IndiaPost	economy	300	210
Ekart	standard	0	0
DTDC	express	150	140

Return `name`, `tier`, `rate` — `100 * on_time / delivered` rounded to 1 decimal, NULL when nothing was delivered — and `grade`: `no data` when the rate is NULL, `excellent` at 95 or above, `ok` at 80 or above, otherwise `poor` (grade on the rounded rate). Order by tier — `express`, `standard`, `economy` — then `name`.

```
name | tier | rate | grade
BlueDart | express | 96 | excellent
DTDC | express | 93.3 | ok
Delhivery | standard | 82 | ok
Ekart | standard | NULL | no data
IndiaPost | economy | 70 | poor
```

INPUT 1

```
CREATE TABLE carriers (name VARCHAR(40),
tier VARCHAR(20), delivered INT, on_time
INT);
INSERT INTO carriers VALUES ('BlueDart',
'express', 200, 192), ('Delhivery',
'standard', 500, 410), ('IndiaPost',
'economy', 300, 210),
  ('Ekart', 'standard', 0, 0), ('DTDC',
'express', 150, 140);
```

OUTPUT 1

```
name | tier | rate | grade
BlueDart | express | 96 | excellent
DTDC | express | 93.3 | ok
Delhivery | standard | 82 | ok
Ekart | standard | NULL | no data
IndiaPost | economy | 70 | poor
```

REFERENCE SOLUTION

```sql
SELECT name,
       tier,
       ROUND(100 * on_time / NULLIF(delivered, 0), 1) AS rate,
       CASE
         WHEN ROUND(100 * on_time / NULLIF(delivered, 0), 1) IS NULL THEN 'no data'
         WHEN ROUND(100 * on_time / NULLIF(delivered, 0), 1) ⩾ 95 THEN 'excellent'
         WHEN ROUND(100 * on_time / NULLIF(delivered, 0), 1) ⩾ 80 THEN 'ok'
         ELSE 'poor'
       END AS grade
FROM carriers
ORDER BY FIELD(tier, 'express', 'standard', 'economy'), name;
```

CHECKPOINT QUESTIONS

1. What does `CONCAT_WS('-', '2024', NULL, '15')` return?

A. `NULL`

B. `'2024--15'`

C. `'2024-15'`

D. `'2024-NULL-15'`

2. What does `SUBSTRING('database', 5, 2)` return?

A. `'ba'`

B. `'as'`

C. `'ab'`

D. `'se'`

3. What is `-7 DIV 2` in MySQL?

A. `-4`

B. `-3`

C. `-3.5`

D. `3`

4. What is `ROUND(-2.5)` in MySQL (for an exact decimal value)?

A. `-2`

B. `-3`

C. `-2.5`

D. `2`

5. What does `MOD(17, 5)` return?

A. `3`

B. `2`

C. `3.4`

D. `12`

6. Which expression turns a date into a key shared by every day of its month, such as `2024-03`?

A. `MONTH(d)`

B. `DATE_FORMAT(d, '%Y-%m')`

C. `DATE_FORMAT(d, '%m-%Y')`

D. `LAST_DAY(d)`

7. Which numbering does `WEEKDAY(d)` use?

A. Sunday = 1 … Saturday = 7

B. Monday = 0 … Sunday = 6

C. Monday = 1 … Sunday = 7

D. Sunday = 0 … Saturday = 6

8. What is `DATE_ADD('2024-01-31', INTERVAL 1 MONTH)`?

A. `'2024-03-02'`

B. `'2024-02-29'`

C. `'2024-02-28'`

D. `'2024-03-01'`

9. Which call returns the number of whole months from `start_date` to `end_date`?

A. `DATEDIFF(MONTH, start_date, end_date)`

B. `TIMESTAMPDIFF(MONTH, start_date, end_date)`

C. `TIMESTAMPDIFF(MONTH, end_date, start_date)`

D. `DATEDIFF(end_date, start_date) / 30`

10. Why write `ROUND(100 * a / NULLIF(b, 0), 1)` instead of `ROUND(100 * a / b, 1)`?

A. `NULLIF` makes the division faster

B. So a zero `b` gives NULL instead of an error in databases that raise one, and the intent is explicit

C. `NULLIF` rounds the result

D. Without it MySQL returns infinity

11. What does `IF(stock > 0, 'yes', 'no')` return when `stock` is NULL?

A. `'yes'`

B. `'no'`

C. `NULL`

D. An error

ANSWER KEY · CHECKPOINT — FUNCTIONS

1. C `CONCAT_WS` skips NULL arguments entirely — no value and no separator for them. Plain `CONCAT` would return NULL.

2. A Positions start at 1: d(1) a(2) t(3) a(4) b(5) a(6). Two characters from position 5 are `ba`.

3. B `DIV` truncates towards zero, so -3.5 becomes -3. `FLOOR(-7 / 2)` would give -4.

4. B For exact (DECIMAL) values MySQL rounds halves away from zero, so -2.5 becomes -3. For floating-point values it may round to even instead — another reason money is DECIMAL.

5. B 5 fits into 17 three times (15) with 2 left over. `17 DIV 5` is the 3; `MOD` is the remainder.

6. B `'%Y-%m'` gives year and month in an order that also sorts correctly. `MONTH(d)` alone merges March of every year; `'%m-%Y'` sorts badly as text; `LAST_DAY` does work as a key but is a date, not `2024-03`.

7. B `WEEKDAY` starts at Monday = 0. `DAYOFWEEK` is the other one, Sunday = 1 through Saturday = 7. Comparing `DAYNAME(d)` avoids mixing them up.

8. B MySQL clamps to the last day of the shorter month, and 2024 is a leap year, so 29 February.

9. B `TIMESTAMPDIFF(unit, start, end)` counts whole units from start to end. The first option is SQL Server's syntax, the third has the dates reversed and gives a negative number, and dividing days by 30 is only approximate.

10. B `NULLIF(b, 0)` turns a zero divisor into NULL, so the result is NULL. MySQL already returns NULL for division by zero, but PostgreSQL and SQL Server raise an error. It has nothing to do with speed or rounding.

11. B `NULL > 0` is unknown, which is not true, so `IF` returns its third argument.