Module 9 of 16

Set operations and CTEs

UNION and UNION ALL for stacking results, INTERSECT and EXCEPT with their NOT EXISTS and anti-join equivalents, the NOT IN null trap, common table expressions for naming the steps of a query, and recursive CTEs for org charts, category paths and date series.

Joins put tables side by side; set operations put result sets on top of each other, and common table expressions put the steps of a query in order. This module covers both. `UNION`, `INTERSECT` and `EXCEPT` answer "rows from either", "rows in both" and "rows in one but not the other" — questions that come up whenever two tables hold the same kind of thing — and `WITH` turns a query that would nest three subqueries deep into a list of named steps that reads top to bottom. `WITH RECURSIVE` then lets SQL follow a link as many times as the data needs, which is how a relational database walks a hierarchy.

UNION and UNION ALL covers the column rules (count, types by position, names from the first SELECT), why `UNION` removes duplicates and what that costs, tagging rows with a constant, sorting the combined result and aggregating over a union. INTERSECT and EXCEPT covers both operations in MySQL 8.0.31+, their portable forms with `EXISTS`, `NOT EXISTS` and `LEFT JOIN … IS NULL`, and the `NOT IN` trap that returns no rows when the list holds a NULL. Common table expressions covers the syntax, chaining, reusing one result twice, and how CTEs compare to derived tables, views and temporary tables. Recursive CTEs covers the anchor and recursive members, walking down and up an org chart, building category paths, generating number and date series, and keeping recursion bounded.

The exercises are queries judged on real tables: a deduplicated mailing list and contact counts per city, students in two courses and products that never sold (with a NULL waiting in the data), departments above the company average and each customer's share of revenue, an org chart by level and full category paths. The checkpoint adds a daily revenue report with the missing days filled in, an activity count across two tables, and every manager's whole team size.

CONTENTS

1	UNION and UNION ALL — stacking result sets	14 min
2	INTERSECT and EXCEPT — rows in both, rows in one but not the other	14 min

3	Common table expressions — naming the steps of a query	15 min
4	Recursive CTEs — hierarchies, paths and number series	16 min
✓	Checkpoint — Set operations and CTEs	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

UNION and UNION ALL — stacking result sets

Joins put tables side by side: each output row is wider because it carries columns from several tables. Set operations go the other way — they put result sets *on top of each other*. The rows of one query and the rows of another become one taller result. `UNION` and `UNION ALL` are the set operations every SQL dialect has had for decades, and they answer a whole family of everyday questions: one mailing list from customers and suppliers, one activity feed from posts and comments, one report from this year's table and the archive table. This lesson covers how a union is formed, the column rules, the difference between `UNION` and `UNION ALL`, sorting the combined result, and the patterns that use them.

Stacking two queries

```
SELECT name, email FROM customers
UNION ALL
SELECT company, email FROM suppliers;
```

Each SELECT runs on its own, with its own FROM, WHERE, GROUP BY and HAVING. Then the second result's rows are appended under the first's. The combined result has the *first* SELECT's column names — here `name` and `email`, even for the rows that came from `suppliers.company`.

name	email
Asha Rao	asha@mail.com
Ben Ortiz	ben@mail.com
Acme Parts	sales@acme.io
Bolt & Co	hi@bolt.co

Three rules make a union legal:

1. **The same number of columns** in every SELECT. MySQL refuses `SELECT a, b … UNION SELECT c …` with "The used SELECT statements have a different number of columns".

2. **Compatible types by position.** The first column of each SELECT becomes one output column, so it should hold the same kind of value. MySQL will quietly convert mismatches (a number under a string column becomes text), which is rarely what you meant.
3. **Names come from the first SELECT.** Aliases in later branches are allowed but ignored. Put the meaningful names on the first branch.

Matching is by *position*, never by name. `SELECT name, email … UNION ALL SELECT email, company …` is legal and wrong: e-mail addresses end up under `name`. When the branches come from tables with different column orders, list the columns explicitly in each branch — never `SELECT *` in a union.

UNION removes duplicates, UNION ALL keeps them

```sql
SELECT city FROM customers
UNION
SELECT city FROM suppliers;
```

`UNION` (short for `UNION DISTINCT`) returns each distinct row once across the *whole* combined result — duplicates within one branch are removed too, not just duplicates between branches. If three customers live in Pune and one supplier does, Pune appears once.

UNION ALL appends; UNION then keeps each distinct row once



1. Each SELECT runs on its own: the customers' cities and the suppliers' cities. Three customers and one supplier are in Pune.
2. UNION ALL appends the second result's rows under the first's and does nothing else: 6 rows, Pune 4 times. No checking, so it streams.
3. UNION must also remember every row it has emitted and test each new one against them. Duplicates are removed across the whole result — the two repeats inside the customers branch too.
4. What remains is the set of distinct cities, Pune, Delhi, Mumbai. Right for "every city we deal with"; wrong for counting or summing, where real repeats matter.

`UNION ALL` keeps every row:

```
SELECT city FROM customers
UNION ALL
SELECT city FROM suppliers;      -- Pune appears four times
```

Which to use is a question about meaning first, cost second:

- When the branches cannot overlap (rows tagged with a different constant, disjoint date ranges, two different tables of events), the duplicate check finds nothing and only costs time. Use `UNION ALL`.
- When you are counting or summing afterwards, `UNION` is usually a bug: two genuinely separate sales of the same amount on the same day look like duplicates and one is thrown away.
- When the question is "the set of distinct values from either source" — every city we deal with, every e-mail to send one message to — `UNION` is exactly right.

The cost difference is real. `UNION ALL` streams rows through. `UNION` must remember every row it has emitted (MySQL builds a temporary table with a unique key) and check each new row against it. On a few thousand rows nobody notices; on millions it is the difference

between a query that streams and one that spills to disk. Interviewers like this question because the answer has both halves: *semantics* (duplicates) and *cost* (the dedup pass).

For deduplication, NULLs count as equal: two rows `('Pune', NULL)` are duplicates to `UNION` even though `NULL = NULL` is not true in a WHERE clause. `DISTINCT` and `GROUP BY` behave the same way.

Tagging where a row came from

Once rows are stacked, you can no longer tell which table each came from — unless you say so. Add a constant column:

```
SELECT name AS contact, email, 'customer' AS kind FROM customers
UNION ALL
SELECT company, email, 'supplier' FROM suppliers;
```

contact	email	kind
Asha Rao	asha@mail.com	customer
Acme Parts	sales@acme.io	supplier

The literal is just another column, so it obeys the same rules: same position in every branch, name from the first. A tag also makes `UNION` and `UNION ALL` give the same answer whenever the tags differ, since no row of one branch can equal a row of the other — another reason `UNION ALL` is the default choice for tagged unions.

Sorting and limiting the combined result

An ORDER BY after the last SELECT sorts the *whole* union, not the last branch:

```
SELECT name AS contact, 'customer' AS kind FROM customers
UNION ALL
SELECT company, 'supplier' FROM suppliers
ORDER BY contact, kind
LIMIT 10;
```

Order by the result's column names — the first branch's names or aliases. Most engines also accept a column position (`ORDER BY 1`), but an expression over the original tables (`ORDER BY customers.name`) is refused: after the union, `customers` no longer exists, only the combined result does.

To sort or limit *one branch* — the three newest posts plus the three newest comments — MySQL lets you parenthesise each SELECT:

```
(SELECT id, created_at, 'post' AS kind FROM posts ORDER BY created_at DESC LI
UNION ALL
(SELECT id, created_at, 'comment' FROM comments ORDER BY created_at DESC LIMI
ORDER BY created_at DESC;
```

Without the LIMIT, an ORDER BY inside a branch is meaningless — the union is free to reorder — and MySQL ignores it. (This course's in-browser engine does not accept parenthesised branches; the same effect is written with a CTE or derived table per branch, which the next lessons cover.)

Aggregating over a union

A union is a result set, so it can be a derived table. To count all activity per user across two tables, union first, then group:

```
SELECT user_id, COUNT(*) AS actions
FROM (
  SELECT user_id FROM posts
  UNION ALL
  SELECT user_id FROM comments
) AS activity
GROUP BY user_id
ORDER BY actions DESC, user_id;
```

`UNION ALL` is essential here: with `UNION`, a user's second post would be "a duplicate" of their first (both rows are just `user_id = 7`) and the count would be wrong. MySQL requires the alias (`AS activity`) on the derived table.

Patterns that use a union

- **Combining similar tables.** Current and archived orders, per-region tables, a legacy table beside its replacement.
- **Turning columns into rows.** A table with `phone_home`, `phone_work`, `phone_mobile` becomes one row per number with three branches: `SELECT id, 'home', phone_home … UNION ALL SELECT id, 'work', phone_work …` (an unpivot, used again in the patterns module).
- **Adding a totals row.** `SELECT region, SUM(amount) … GROUP BY region UNION ALL SELECT 'Total', SUM(amount) FROM sales`. MySQL also has `GROUP BY … WITH ROLLUP` for this; the union version works everywhere and lets you control the label.
- **Replacing an OR across two indexed columns.** `WHERE a = 1 OR b = 2` can stop an index being used; two branches each using its own index, joined by `UNION`, sometimes run far faster. Here `UNION` (not ALL) is needed, because a row matching both conditions would otherwise appear twice.

In other databases

PostgreSQL, SQL Server, Oracle and SQLite all have `UNION` and `UNION ALL` with the same meaning and the same column rules. Oracle is strict about types and refuses a mismatch MySQL would convert. SQL Server has no `LIMIT`; it uses `TOP` or `OFFSET … FETCH` on the final query. The duplicate-versus-cost answer is identical everywhere.

Pitfalls

- Using `UNION` where duplicates are real data, then wondering why a sum is short.
- Matching columns by name instead of position — the second branch's columns land under the wrong headings without an error.
- Putting the aliases on the second branch and wondering why the result uses the first branch's raw names.
- Ordering by a table-qualified column after a union.
- Forgetting the alias on a derived table that wraps a union.

Key takeaways

- A union stacks rows: same column count, compatible types by position, names from the first SELECT.
- `UNION` removes duplicates across the whole result (NULLs count as equal); `UNION ALL` keeps every row and is cheaper.
- Default to `UNION ALL`; reach for `UNION` when "distinct values from either source" is the actual question.
- One ORDER BY at the end sorts the combined result; tag rows with a constant column to keep their origin.
- Wrap a union in a derived table to group or count across both sources.

EXERCISE 1

A deduplicated mailing list

Marketing wants one mailing list of everyone in `customers` and `suppliers`. An import loaded some customers twice, so each distinct contact must appear **once**.

customers: id	name	email	city
1	Asha Rao	asha@mail.com	Pune
2	Ben Ortiz	ben@mail.com	Delhi
3	Asha Rao	asha@mail.com	Pune
4	Chen Li	chen@mail.com	Pune

suppliers: id	company	email	city
1	Acme Parts	sales@acme.io	Mumbai
2	Bolt and Co	hi@bolt.co	Pune

Return the columns `contact` (a customer's `name` or a supplier's `company`), `email` and `kind` (`'customer'` or `'supplier'`), each distinct row once, ordered by `contact`, then `kind`.

```
contact | email | kind
Acme Parts | sales@acme.io | supplier
Asha Rao | asha@mail.com | customer
Ben Ortiz | ben@mail.com | customer
Bolt and Co | hi@bolt.co | supplier
Chen Li | chen@mail.com | customer
```

Someone who is both a customer and a supplier appears twice — once per `kind`.

INPUT 1

```
CREATE TABLE customers (id INT PRIMARY
KEY, name VARCHAR(50), email
VARCHAR(80), city VARCHAR(30));
CREATE TABLE suppliers (id INT PRIMARY
KEY, company VARCHAR(50), email
VARCHAR(80), city VARCHAR(30));
INSERT INTO customers VALUES
  (1, 'Asha Rao', 'asha@mail.com',
'Pune'),
  (2, 'Ben Ortiz', 'ben@mail.com',
'Delhi'),
  (3, 'Asha Rao', 'asha@mail.com',
'Pune'),
  (4, 'Chen Li', 'chen@mail.com',
'Pune');
INSERT INTO suppliers VALUES
  (1, 'Acme Parts', 'sales@acme.io',
'Mumbai'),
  (2, 'Bolt and Co', 'hi@bolt.co',
'Pune');
```

OUTPUT 1

```
contact | email | kind
Acme Parts | sales@acme.io | supplier
Asha Rao | asha@mail.com | customer
Ben Ortiz | ben@mail.com | customer
Bolt and Co | hi@bolt.co | supplier
Chen Li | chen@mail.com | customer
```

REFERENCE SOLUTION

```sql
SELECT name AS contact, email, 'customer' AS kind FROM customers
UNION
SELECT company, email, 'supplier' FROM suppliers
ORDER BY contact, kind;
```

EXERCISE 2

Contacts per city

Count every contact — customers and suppliers together — per city. Every row is a real contact here, so nothing may be collapsed; rows whose `city` is NULL are left out.

customers: id	name	email	city
1	Asha Rao	asha@mail.com	Pune
2	Ben Ortiz	ben@mail.com	Delhi
3	Chen Li	chen@mail.com	Pune
4	Gita Nair	gita@mail.com	Chennai

suppliers: id	company	email	city
1	Acme Parts	sales@acme.io	Mumbai
2	Bolt and Co	hi@bolt.co	Pune
3	Delta Foods	ops@delta.in	Delhi

Return `city` and `contacts`, ordered by `contacts` descending, then `city`.

```
city | contacts
Pune | 3
Delhi | 2
Chennai | 1
Mumbai | 1
```

INPUT 1

```
CREATE TABLE customers (id INT PRIMARY
KEY, name VARCHAR(50), email
VARCHAR(80), city VARCHAR(30));
CREATE TABLE suppliers (id INT PRIMARY
KEY, company VARCHAR(50), email
VARCHAR(80), city VARCHAR(30));
INSERT INTO customers VALUES
  (1, 'Asha Rao', 'asha@mail.com',
'Pune'),
  (2, 'Ben Ortiz', 'ben@mail.com',
'Delhi'),
  (3, 'Chen Li', 'chen@mail.com',
'Pune'),
  (4, 'Gita Nair', 'gita@mail.com',
'Chennai');
INSERT INTO suppliers VALUES
  (1, 'Acme Parts', 'sales@acme.io',
'Mumbai'),
  (2, 'Bolt and Co', 'hi@bolt.co',
'Pune'),
  (3, 'Delta Foods', 'ops@delta.in',
'Delhi');
```

OUTPUT 1

```
city | contacts
Pune | 3
Delhi | 2
Chennai | 1
Mumbai | 1
```

REFERENCE SOLUTION

```sql
SELECT city, COUNT(*) AS contacts
FROM (
  SELECT city FROM customers
  UNION ALL
  SELECT city FROM suppliers
) AS everyone
WHERE city IS NOT NULL
GROUP BY city
ORDER BY contacts DESC, city;
```

QUIZ

1. Which column names does this result have?

```sql
SELECT name, email FROM customers
UNION ALL
SELECT company AS name2, email AS mail FROM suppliers;
```

A. `name`, `email`

B. `name2`, `mail`

C. `name`, `mail`

D. It is an error: the names differ

2. `customers.city` holds Pune, Pune, Delhi and `suppliers.city` holds Pune. How many rows does `SELECT city FROM customers UNION SELECT city FROM suppliers` return?

A. 4

B. 3

C. 2

D. 1

3. You sum the `amount` of `sales_2023 UNION sales_2024` (both with columns `day, amount`) and the total is lower than expected. Why?

A. UNION skips NULL amounts

B. UNION removed rows that were identical, such as two equal sales on the same day

C. UNION sorts the rows and drops the last one

D. SUM cannot read a union

4. Where does an ORDER BY written after the last SELECT of a union apply?

A. To the last SELECT only

B. To the whole combined result

C. To the first SELECT only

D. Nowhere — it is ignored

5. Why is `UNION ALL` usually faster than `UNION`?

A. It uses indexes and UNION does not

B. It skips the duplicate-removal pass over the combined rows

C. It runs the branches in parallel

D. It returns fewer rows

ANSWER KEY · UNION AND UNION ALL — STACKING RESULT SETS

1. A A union takes its column names from the first SELECT; aliases in later branches are ignored. Different names are not an error — columns are matched by position, which is also why listing them in a different order silently mixes them up.

2. C `UNION` removes duplicates across the whole combined result, including duplicates within one branch: Pune and Delhi, two rows. `UNION ALL` would return all 4.

3. B Two genuine sales with the same day and amount are duplicate *rows*, and `UNION` keeps one. When every row is real data, use `UNION ALL`. NULLs are not skipped by a union, and a union can be summed through a derived table.

4. B A trailing ORDER BY (and LIMIT) sorts the combined result, using the first SELECT's column names. To sort or limit one branch alone, MySQL needs that SELECT in parentheses with its own LIMIT.

5. B `UNION` has to remember every row emitted (MySQL uses a temporary table with a unique key) and check each new one against it. `UNION ALL` just appends. It returns as many rows or more, never fewer.

LESSON 2 · 14 MIN

INTERSECT and EXCEPT — rows in both, rows in one but not the other

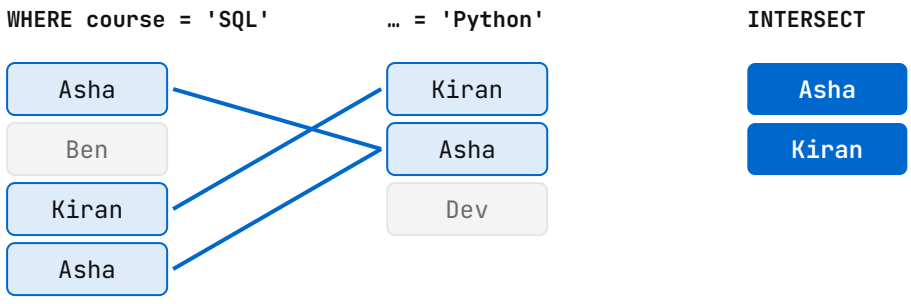
`UNION` answers "rows from either query". Two more set operations finish the family: `INTERSECT` answers "rows in *both*", and `EXCEPT` answers "rows in the first but *not* the second". Students enrolled in both SQL and Python; products in the catalogue that never sold; customers who ordered last year but not this year — each is one of these two operations. MySQL only gained them in version 8.0.31, so production code and interview answers are full of the older ways to write the same thing: `EXISTS`, `NOT EXISTS` and joins. This lesson teaches both forms, and why one popular form, `NOT IN`, is a trap.

INTERSECT

```sql
SELECT student FROM enrollments WHERE course = 'SQL'
INTERSECT
SELECT student FROM enrollments WHERE course = 'Python';
```

Each SELECT runs on its own; the result is every row that appears in both results, once. The rules are the union rules: the same number of columns, compatible types by position, names from the first branch, one ORDER BY at the end that sorts the final result.

INTERSECT: the rows both results contain, once each



1. Each SELECT runs on its own: the students taking SQL and the students taking Python. Asha is enrolled in SQL twice.
2. INTERSECT compares whole rows across the two results. Asha and Kiran appear in both; Ben and Dev appear in only one.
3. The result is each shared row once — Asha, Kiran — Asha's two SQL rows included. Like UNION, INTERSECT removes duplicates.

`INTERSECT` compares *whole rows*. Intersecting `(student, city)` pairs requires the same student in the same city in both results — not just the same student. And like `UNION`, it removes duplicates: a student enrolled in SQL twice still appears once. The standard also defines `INTERSECT ALL` (keep `min(m, n)` copies of a row appearing m and n times), which MySQL 8.0.31+ and PostgreSQL support; it is rarely needed.

EXCEPT

```sql
SELECT sku FROM products
EXCEPT
SELECT sku FROM sales;
```

Every distinct row of the first result that does not appear in the second: the products that never sold. Order matters — `sales EXCEPT products` would be "SKUs sold that are not in the catalogue", a data-quality check rather than a business question. Oracle spells it `MINUS`.

Both operations treat NULLs as equal when comparing rows, exactly like `DISTINCT`. That makes them safer than you might expect, as the NOT IN section will show.

Precedence and mixing

When one statement mixes the operators, `INTERSECT` binds tighter than `UNION` and `EXCEPT` in the SQL standard, MySQL, PostgreSQL and SQL Server:

```sql
SELECT id FROM a
UNION
SELECT id FROM b
INTERSECT
SELECT id FROM c;          -- a UNION (b INTERSECT c)
```

SQLite reads the same statement left to right. Rather than remember, make the grouping explicit with parentheses (in MySQL) or with a CTE per step (next lesson).

The emulation: EXISTS and NOT EXISTS

Before 8.0.31 — and in a great deal of code still running — the two operations are written as semi-joins and anti-joins. The intersection:

```
SELECT DISTINCT e.student
FROM enrollments e
WHERE e.course = 'SQL'
  AND EXISTS (SELECT 1
              FROM enrollments p
              WHERE p.student = e.student
                AND p.course = 'Python');
```

And the difference:

```
SELECT p.sku
FROM products p
WHERE NOT EXISTS (SELECT 1 FROM sales s WHERE s.sku = p.sku);
```

The correlated subquery asks, per outer row, "is there a matching row over there?" `EXISTS` keeps the row when the answer is yes, `NOT EXISTS` when it is no. The optimizer turns both into a semi-join or anti-join that stops at the first match. Two details matter:

- **Duplicates.** `INTERSECT` and `EXCEPT` return distinct rows; the EXISTS forms return the outer table's rows as they are. Add `DISTINCT` if the outer side can repeat (a student enrolled in SQL twice).
- **Multi-column rows.** Compare every column in the correlation: `WHERE s.sku = p.sku AND s.store = p.store`. A NULL in a compared column never matches with `=` — where `EXCEPT` would call two NULL rows equal, NOT EXISTS keeps the row. Usually that difference does not arise, because keys are not NULL.

The same anti-join is often written as a LEFT JOIN that finds no partner:

```
SELECT p.sku
FROM products p
LEFT JOIN sales s ON s.sku = p.sku
WHERE s.sku IS NULL;
```

The LEFT JOIN keeps every product and fills the sales columns with NULL when no sale matches; `WHERE s.sku IS NULL` keeps exactly those. It returns one row per product only because unmatched products produce a single NULL-extended row — matched products produce one row per sale and are then all filtered out. Test the *join column* (or a NOT NULL column) of the right table for NULL, never a nullable column such as an amount.

An intersection can also be written as an inner join — `SELECT DISTINCT a.student FROM enrollments a JOIN enrollments b ON b.student = a.student WHERE a.course = 'SQL' AND b.course = 'Python'` — or with `GROUP BY student HAVING COUNT(DISTINCT course) = 2` over `WHERE course IN ('SQL', 'Python')`. The last form generalises nicely: "enrolled in all of these five courses" is `= 5`.

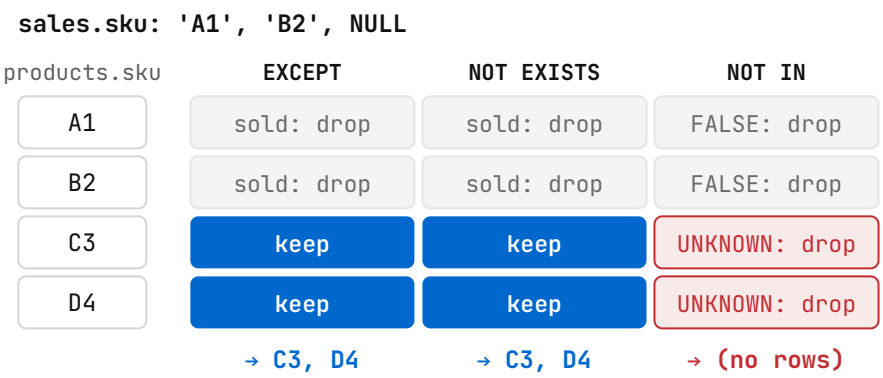
The NOT IN trap

The tempting third way to write "never sold":

```sql
SELECT sku FROM products
WHERE sku NOT IN (SELECT sku FROM sales);
```

It works — until one row in `sales` has `sku` NULL. Then it returns *nothing*. `x NOT IN (a, b, NULL)` expands to `x <> a AND x <> b AND x <> NULL`; the last comparison is unknown, so the conjunction is never true, so no row passes. No error, no warning, an empty result that looks plausible on a busy day.

Products never sold: EXCEPT, NOT EXISTS and NOT IN



1. Four products and three sales, one of them with a NULL sku. The question: which products never sold? The answer should be C3 and D4.
2. EXCEPT removes every product row that also appears in sales. It compares whole rows with NULLs treated as equal, so the NULL sale is just a row matching nothing here: C3 and D4 remain.
3. NOT EXISTS asks, per product, whether some sale has s.sku = p.sku. For C3 the NULL sale gives UNKNOWN, which is not a match, so no matching row exists: C3 and D4 again.
4. NOT IN expands to C3 <> 'A1' AND C3 <> 'B2' AND C3 <> NULL. The last test is UNKNOWN, so the AND never becomes TRUE and the query returns no rows at all.

Defences, in order of preference: use `NOT EXISTS` or `EXCEPT`; or filter the subquery (`WHERE sku IS NOT NULL`); or make the column `NOT NULL` in the schema so the case cannot arise.

Interviewers ask this exact question, and "NOT IN with a NULL returns no rows" is the answer they are waiting for. (`IN` is not affected the same way: `x IN (a, NULL)` is true when `x = a`, and merely unknown otherwise, which filters the same as false.)

Choosing a form

Need	Modern	Portable
rows in both	`INTERSECT`	`EXISTS`, inner join + `DISTINCT`
rows in A not B	`EXCEPT`	`NOT EXISTS`, `LEFT JOIN … IS NULL`
whole-row comparison with NULLs equal	`INTERSECT` / `EXCEPT`	`NOT EXISTS` with `⟺` per column

`INTERSECT` and `EXCEPT` read best when both sides are naturally "a list of the same kind of thing". When you need columns from the outer table that the other side does not have — the product's *name*, not just its SKU — the EXISTS forms are simpler, because a set operation can only return columns both sides have. You can also combine them: compute the SKUs with `EXCEPT` in a CTE, then join back to `products` for the names.

On performance, modern optimizers usually turn NOT EXISTS and LEFT JOIN … IS NULL into the same anti-join plan; `EXCEPT` may instead build a temporary table of the second result. Measure on real data when it matters (the indexes module shows how).

In other databases

PostgreSQL, SQL Server and SQLite have had `INTERSECT` and `EXCEPT` for years; Oracle uses `MINUS` (and accepts `EXCEPT` from 21c). SQL Server's `EXCEPT` and `INTERSECT` have no `ALL` form; PostgreSQL's do. The NOT IN null trap is identical in every one of them, because it comes from three-valued logic, not from any product.

Pitfalls

- `NOT IN` against a subquery whose column can be NULL.
- Swapping the branches of `EXCEPT`.

- Forgetting that the EXISTS forms keep duplicates the set operations would remove.
- Testing a nullable column for NULL after a LEFT JOIN anti-join.
- Mixing operators without parentheses and relying on precedence.

Key takeaways

- `INTERSECT` keeps distinct rows present in both results; `EXCEPT` keeps distinct rows of the first missing from the second.
- MySQL has both from 8.0.31; `EXISTS`, `NOT EXISTS` and `LEFT JOIN … IS NULL` express them on any version.
- Set operations compare whole rows by position and treat NULLs as equal.
- `NOT IN` returns no rows if the list contains a NULL — prefer `NOT EXISTS` or `EXCEPT`.
- Use a set operation for "same kind of list"; use EXISTS when you need the outer table's other columns.

EXERCISE 1

Students in both courses

Find the students enrolled in **both** `'SQL'` and `'Python'`. A student may appear more than once for the same course.

enrollments: student	course
Asha	SQL
Asha	Python
Ben	SQL
Kiran	Python
Kiran	SQL
Kiran	SQL
Meera	Java
Meera	Python

Return one column, `student`, each student once, ordered by `student`.

```
student
Asha
Kiran
```

`INTERSECT`, `EXISTS` and `GROUP BY … HAVING` all work — try more than one.

INPUT 1

```
CREATE TABLE enrollments (student
VARCHAR(30), course VARCHAR(30));
INSERT INTO enrollments VALUES
  ('Asha', 'SQL'), ('Asha', 'Python'),
('Ben', 'SQL'), ('Kiran', 'Python'),
  ('Kiran', 'SQL'), ('Kiran', 'SQL'),
('Meera', 'Java'), ('Meera', 'Python');
```

OUTPUT 1

```
student
Asha
Kiran
```

REFERENCE SOLUTION

```sql
SELECT student FROM enrollments WHERE course = 'SQL'
INTERSECT
SELECT student FROM enrollments WHERE course = 'Python'
ORDER BY student;
```

EXERCISE 2

Products that never sold

List the products with no row in `sales`. The `sales.sku` column is nullable — a till sometimes records a sale without a SKU.

products: sku	name	price
A1	Kettle	1499.00
B2	Toaster	1999.00
C3	Blender	2499.00
D4	Mixer	3299.00

sales: id	sku	qty
1	A1	2
2	B2	1
3	A1	1

Return `sku` and `name`, ordered by `sku`.

```
sku | name
C3 | Blender
D4 | Mixer
```

Make sure your query still works when `sales` contains a row whose `sku` is NULL.

INPUT 1

```
CREATE TABLE products (sku VARCHAR(10)
PRIMARY KEY, name VARCHAR(40), price
DECIMAL(8,2));
CREATE TABLE sales (id INT PRIMARY KEY,
sku VARCHAR(10), qty INT);
INSERT INTO products VALUES
  ('A1', 'Kettle', 1499.00), ('B2',
'Toaster', 1999.00), ('C3', 'Blender',
2499.00), ('D4', 'Mixer', 3299.00);
INSERT INTO sales VALUES (1, 'A1', 2),
(2, 'B2', 1), (3, 'A1', 1);
```

OUTPUT 1

```
sku | name
C3 | Blender
D4 | Mixer
```

REFERENCE SOLUTION

```sql
SELECT p.sku, p.name
FROM products p
WHERE NOT EXISTS (SELECT 1 FROM sales s WHERE s.sku = p.sku)
ORDER BY p.sku;
```

QUIZ

1. `a.id` holds 1, 2, 3 and `b.id` holds 2, NULL. What does `SELECT id FROM a WHERE id NOT IN (SELECT id FROM b)` return?

A. 1 and 3

B. 1, 3 and NULL

C. No rows

D. An error

2. Which MySQL version first accepts `INTERSECT` and `EXCEPT`?

A. 5.7

B. 8.0.0

C. 8.0.31

D. They are not supported; MySQL only has UNION

3. Which query returns the same rows as `SELECT sku FROM products EXCEPT SELECT sku FROM sales` when `sku` is never NULL?

A. `SELECT p.sku FROM products p JOIN sales s ON s.sku = p.sku`

B. `SELECT p.sku FROM products p LEFT JOIN sales s ON s.sku = p.sku WHERE s.sku IS NULL`

C. `SELECT p.sku FROM products p LEFT JOIN sales s ON s.sku = p.sku WHERE p.sku IS NULL`

D. `SELECT sku FROM sales WHERE sku NOT IN (SELECT sku FROM products)`

4. Why might `SELECT e.student FROM enrollments e WHERE e.course = 'SQL' AND EXISTS (…Python…)` return more rows than the equivalent `INTERSECT`?

A. EXISTS ignores the WHERE clause

B. EXISTS keeps the outer table's duplicate rows, while INTERSECT returns distinct rows

C. INTERSECT drops students with NULL courses

D. It cannot — they always return the same rows

5. Oracle's name for `EXCEPT` is…

A. `DIFFERENCE`

B. `MINUS`

C. `SUBTRACT`

D. `NOT UNION`

ANSWER KEY · INTERSECT AND EXCEPT — ROWS IN BOTH, ROWS IN ONE BUT NOT THE OTHER

1. C `id NOT IN (2, NULL)` means `id <> 2 AND id <> NULL`; the second comparison is unknown, so the condition is never true. `NOT EXISTS` or `SELECT id FROM a EXCEPT SELECT id FROM b` return 1 and 3.

2. C MySQL 8.0.31 (October 2022) added both. Earlier versions, including the first 8.0 releases, only have UNION — which is why the EXISTS forms are so common.

3. B The LEFT JOIN keeps every product and fills sales columns with NULL where nothing matched; testing the right table's join column for NULL keeps exactly the unmatched products. The inner join gives the sold ones, testing `p.sku` finds nothing, and the last query reverses the direction.

4. B A student enrolled in SQL twice passes the EXISTS test twice. INTERSECT, like UNION, returns distinct rows. Add DISTINCT to the EXISTS form to match.

5. B Oracle has long used `MINUS`; Oracle 21c also accepts `EXCEPT`. The SQL standard, PostgreSQL, SQL Server, SQLite and MySQL 8.0.31+ use `EXCEPT`.

LESSON 3 · 15 MIN

Common table expressions — naming the steps of a query

A query that answers a real business question is rarely one SELECT. "Which departments pay above the company average, and by how much?" needs the average per department, the company average, and then a comparison — three steps. Written with derived tables, the steps nest inside one another and the query reads inside out. A **common table expression** (CTE) gives each step a name at the top of the statement, so the query reads top to bottom like a recipe. CTEs are standard SQL, supported by MySQL since 8.0, and the shape most interviewers expect a multi-step answer in. This lesson covers the syntax, chaining, reuse, how CTEs compare to subqueries, views and temporary tables, and the habits that keep them readable.

The syntax

```
WITH dept_avg AS (
  SELECT dept, AVG(salary) AS avg_salary
  FROM employees
  GROUP BY dept
)
SELECT dept, ROUND(avg_salary, 2) AS avg_salary
FROM dept_avg
WHERE avg_salary > 60000
ORDER BY avg_salary DESC, dept;
```

`WITH name AS ( query )` defines the CTE; the statement that follows reads `name` exactly like a table — in FROM, in a join, in a subquery. The CTE exists only for this one statement: the next query cannot see `dept_avg`, and nothing is stored.

You can name the columns in a list instead of with aliases inside:

```
WITH dept_avg (dept, avg_salary) AS (
  SELECT dept, AVG(salary) FROM employees GROUP BY dept
)
SELECT * FROM dept_avg;
```

Both are fine; the inner aliases keep each name beside its expression and are the more common style.

Chaining CTEs

Write `WITH` once and separate the definitions with commas. Each CTE may read every CTE defined *before* it:

```sql
WITH dept_avg AS (
  SELECT dept, AVG(salary) AS avg_salary
  FROM employees
  GROUP BY dept
),
company AS (
  SELECT AVG(salary) AS avg_salary FROM employees
)
SELECT d.dept,
       ROUND(d.avg_salary, 2) AS avg_salary,
       ROUND(d.avg_salary - c.avg_salary, 2) AS above_company
FROM dept_avg d
CROSS JOIN company c
WHERE d.avg_salary > c.avg_salary
ORDER BY above_company DESC, d.dept;
```

`company` has exactly one row, so the cross join attaches the company average to every department row — a common, deliberate use of CROSS JOIN. Read the statement top to bottom: averages per department, the company average, then the comparison. Each step can be tested alone by running its SELECT, which is the most practical benefit of the style.

A longer pipeline follows the same pattern: clean the raw rows, aggregate them, rank or filter, then present.

```sql
WITH paid AS (                -- step 1: the rows that count
  SELECT customer_id, amount FROM orders WHERE status = 'paid'
),
totals AS (                   -- step 2: one row per customer
  SELECT customer_id, SUM(amount) AS total FROM paid GROUP BY customer_id
),
grand AS (                    -- step 3: one number
  SELECT SUM(total) AS grand_total FROM totals
)
SELECT t.customer_id,
       t.total,
       ROUND(100 * t.total / g.grand_total, 1) AS pct
FROM totals t CROSS JOIN grand g
ORDER BY t.total DESC, t.customer_id;
```

Reusing one CTE

A derived table is written where it is used; to use the same result twice you must write it twice. A CTE is written once and referenced as often as needed:

```sql
WITH monthly AS (
  SELECT DATE_FORMAT(order_date, '%Y-%m') AS month, SUM(amount) AS revenue
  FROM orders
  GROUP BY DATE_FORMAT(order_date, '%Y-%m')
)
SELECT cur.month, cur.revenue, prev.revenue AS prev_revenue
FROM monthly cur
LEFT JOIN monthly prev
  ON prev.month = DATE_FORMAT(DATE_SUB(CONCAT(cur.month, '-01'), INTERVAL 1 M(
ORDER BY cur.month;
```

Here `monthly` is joined to itself to put each month beside the previous one. (The window-functions module does this more simply with `LAG`; the self-join of a CTE is the classic pre-window answer and still common.) When a CTE is referenced more than once, MySQL may *materialize* it — compute it once into an internal temporary table — rather than repeat the work.

CTEs versus subqueries, views and temporary tables

	Lives for	Reusable in the statement	Stored	Indexable
Derived table (subquery in FROM)	the statement	no — repeat it	no	no
CTE	the statement	yes, by name	no	no
View	until dropped	yes, in any statement	the definition only	no (uses base indexes)
Temporary table	the session	yes, in any statement	the rows	yes

Performance. A CTE is not automatically faster or slower than the equivalent derived table. MySQL's optimizer usually *merges* a CTE referenced once into the outer query, so predicates are pushed down and indexes on the base tables are used exactly as if you had written one flat query. A CTE referenced several times may be materialized once. PostgreSQL before version 12 always materialized CTEs (an "optimization fence" — filters outside did not reach inside), which is the origin of the folklore that CTEs are slow; since 12 it inlines them too, unless you write `AS MATERIALIZED`. SQL Server never materializes a CTE: it re-expands it every time it is referenced, so a CTE used three times may run three times.

When to use which. CTEs for structure inside one query. A view when many queries need the same definition. A temporary table when a heavy intermediate result is read by several statements or needs an index.

Rules worth knowing

- **Scope.** A CTE is visible to the statement it belongs to, including subqueries inside it. CTEs defined inside a subquery are visible only there.
- **Names shadow tables.** A CTE named `orders` hides the real `orders` table for that statement. Avoid it; it confuses every reader.
- **Not in the CTE's own body.** A plain CTE cannot refer to itself — that is what `WITH RECURSIVE` (next lesson) is for.
- **With data changes.** In MySQL, `WITH` may precede `UPDATE` and `DELETE` (and the SELECT part of `INSERT … SELECT`), so the same named steps can drive a change. PostgreSQL

goes further, with data-modifying CTEs (`WITH moved AS (DELETE … RETURNING *) INSERT …`); MySQL has no equivalent.

- **ORDER BY inside a CTE is not an order.** The final SELECT decides the row order; always end with an ORDER BY that fixes it completely.

Readability habits

Name each CTE after what its rows *are* (`dept_avg`, `paid_orders`, `first_purchase`), not `t1`, `cte2`. Keep each step to one idea: filter, or aggregate, or join — not all three. Put a short comment on the step whose purpose is not obvious from its name. In an interview, building the answer as a chain of CTEs lets you talk through each step and test it, and it is far easier to fix one step than to unpick a five-level nested subquery.

Pitfalls

- Expecting a CTE to persist into the next statement.
- Referring to a later CTE from an earlier one — order of definition matters.
- Assuming the CTE's ORDER BY orders the result.
- Assuming a CTE is cached or faster without checking the plan.
- Writing `WITH` twice instead of one `WITH` with commas.

Key takeaways

- `WITH name AS (query)` names a result set for one statement; the main query reads it like a table.
- Chain CTEs with commas; each can read the ones defined before it, so a query becomes a list of named steps.
- A CTE can be referenced several times, which a derived table cannot without being repeated.
- CTEs are about readability: MySQL usually merges them like derived tables, and may materialize one used repeatedly.
- Use views for definitions shared across queries and temporary tables for results shared across statements.

EXERCISE 1

Departments above the company average

Which departments pay more on average than the company as a whole, and by how much?

employees: id	name	dept	salary
1	Asha	Eng	90000
2	Ben	Eng	70000
3	Chen	Sales	50000
4	Dev	Sales	40000
5	Eva	HR	45000
6	Farah	Data	85000

Return `dept`, `avg_salary` (the department's average, `ROUND`ed to 2 places) and `above_company` (department average minus the average of every employee, `ROUND`ed to 2 places), only for departments whose average is strictly greater than the company average. Order by `above_company` descending, then `dept`.

```
dept | avg_salary | above_company
Data | 85000 | 21666.67
Eng | 80000 | 16666.67
```

Write it as two CTEs: one row per department, and one row for the company.

INPUT 1

```
CREATE TABLE employees (id INT PRIMARY
KEY, name VARCHAR(30), dept VARCHAR(20),
salary INT);
INSERT INTO employees VALUES
  (1, 'Asha', 'Eng', 90000), (2, 'Ben',
'Eng', 70000), (3, 'Chen', 'Sales',
50000),
  (4, 'Dev', 'Sales', 40000), (5, 'Eva',
'HR', 45000), (6, 'Farah', 'Data',
85000);
```

OUTPUT 1

```
dept | avg_salary | above_company
Data | 85000 | 21666.67
Eng | 80000 | 16666.67
```

REFERENCE SOLUTION

```sql
WITH dept_avg AS (
  SELECT dept, AVG(salary) AS avg_salary
  FROM employees
  GROUP BY dept
),
company AS (
  SELECT AVG(salary) AS avg_salary FROM employees
)
SELECT d.dept,
       ROUND(d.avg_salary, 2) AS avg_salary,
       ROUND(d.avg_salary - c.avg_salary, 2) AS above_company
FROM dept_avg d
CROSS JOIN company c
WHERE d.avg_salary > c.avg_salary
ORDER BY above_company DESC, d.dept;
```

EXERCISE 2

Each customer's share of revenue

Using only orders whose `status` is `'paid'`, show each customer's paid total and its share of all paid revenue.

orders: id	customer	amount	status
1	Asha	120	paid
2	Ben	80	paid
3	Asha	30	paid
4	Chen	200	refunded
5	Chen	50	paid
6	Ben	70	paid

Return `customer`, `total` (sum of their paid amounts) and `pct` (`100 * total / all paid revenue`, `ROUND`ed to 1 place). Customers with no paid order do not appear. Order by `total` descending, then `customer`.

```
customer | total | pct
Asha | 150 | 42.9
Ben | 150 | 42.9
Chen | 50 | 14.3
```

INPUT 1

```
CREATE TABLE orders (id INT PRIMARY KEY,
customer VARCHAR(30), amount
DECIMAL(10,2), status VARCHAR(10));
INSERT INTO orders VALUES
  (1, 'Asha', 120, 'paid'), (2, 'Ben',
80, 'paid'), (3, 'Asha', 30, 'paid'),
  (4, 'Chen', 200, 'refunded'), (5,
'Chen', 50, 'paid'), (6, 'Ben', 70,
'paid');
```

OUTPUT 1

```
customer | total | pct
Asha | 150 | 42.9
Ben | 150 | 42.9
Chen | 50 | 14.3
```

REFERENCE SOLUTION

```sql
WITH paid AS (
  SELECT customer, amount FROM orders WHERE status = 'paid'
),
totals AS (
  SELECT customer, SUM(amount) AS total FROM paid GROUP BY customer
),
grand AS (
  SELECT SUM(total) AS grand_total FROM totals
)
SELECT t.customer, t.total, ROUND(100 * t.total / g.grand_total, 1) AS pct
FROM totals t
CROSS JOIN grand g
ORDER BY t.total DESC, t.customer;
```

QUIZ

1. What is wrong with this statement?

```sql
WITH a AS (SELECT 1 AS x)
WITH b AS (SELECT x + 1 AS y FROM a)
SELECT y FROM b;
```

A. Nothing

B. `WITH` is written once; further CTEs follow after commas

C. A CTE cannot read another CTE

D. CTEs must be named with a column list

2. A CTE is defined in one SELECT. Can the next statement in the same session read it?

A. Yes, until the session ends

B. Yes, until the next COMMIT

C. No — it exists only for the statement that defines it

D. Only if it was materialized

3. What can a CTE do that a derived table (a subquery in FROM) cannot?

A. Use GROUP BY

B. Be referenced more than once in the statement without being written out again

C. Be joined to another table

D. Use an index

4. Is a query rewritten with CTEs faster than the same query with derived tables in MySQL?

A. Always — CTEs are cached

B. Never — CTEs are always materialized

C. Not by itself: MySQL usually merges a CTE like a derived table, and may materialize one that is referenced several times

D. Only on MySQL 5.7

ANSWER KEY · COMMON TABLE EXPRESSIONS — NAMING THE STEPS OF A QUERY

1. B The form is `WITH a AS (…), b AS (…) SELECT …`. `b` may read `a` because `a` is defined before it. Column lists are optional.

2. C A CTE is part of one statement. To keep a result for several statements use a temporary table; to keep a definition use a view.

3. B A derived table is written where it is used, so using it twice means repeating it. A CTE is named once and referenced as often as needed — including from itself, with RECURSIVE. Both can group and be joined.

4. C CTEs are mainly about readability. The optimizer merges a single-use CTE into the outer query, so the plan is often identical. MySQL 5.7 has no CTEs at all. The "CTEs are slow" folklore comes from PostgreSQL before version 12, which always materialized them.

LESSON 4 · 16 MIN

Recursive CTEs — hierarchies, paths and number series

Some data refers to itself. An employee row has a `manager_id` that is another employee's `id`; a category has a `parent_id` that is another category; a flight connects to more flights. Ordinary SQL can follow such a link a fixed number of times — one self-join per level — but not "as many times as the data needs". A **recursive CTE** can. It repeats a query over its own output until nothing new appears, which is how SQL walks an org chart of unknown depth, builds the full path of a category, finds every part inside an assembly, or simply generates the numbers 1 to 100 or every date in a month. This lesson covers how the repetition works, the classic hierarchy queries, building paths, generating series, and how to keep recursion bounded.

The shape

```
WITH RECURSIVE nums AS (
  SELECT 1 AS n                 -- anchor member
  UNION ALL
  SELECT n + 1 FROM nums        -- recursive member
  WHERE n < 5
)
SELECT n FROM nums ORDER BY n;
```

n
1
2
3
4
5

Evaluation goes in rounds:

1. The **anchor** runs once. Its rows (`n = 1`) are the first round, and they go into the result.

2. The **recursive member** runs with `nums` meaning *only the previous round's rows*. From `n = 1` it makes `n = 2` — round two.
3. Step 2 repeats on each new round: 2 makes 3, 3 makes 4, 4 makes 5. From 5, `WHERE n < 5` filters everything out — the round is empty and recursion stops.
4. The result is the union of all rounds.

In MySQL the word `RECURSIVE` is required if any CTE in the `WITH` list refers to itself (PostgreSQL requires it too; SQL Server and Oracle have no keyword and detect self-reference). The two members are joined with `UNION ALL`; plain `UNION` is allowed and removes rows already produced, which can stop a cycle but costs a duplicate check every round.

The recursive member may not use aggregates, window functions, `GROUP BY`, `ORDER BY`, `DISTINCT` or `LIMIT` in MySQL (`LIMIT` is allowed from 8.0.19), and must refer to the CTE exactly once, in its FROM clause — not in a subquery. All of these exist to keep the "one round at a time" model well defined. Aggregate in the final SELECT instead.

Walking down an org chart

`employees`

id	name	manager_id
1	Meera	NULL
2	Arjun	1
3	Divya	1
4	Kabir	2
5	Leela	4

Everyone under the CEO, with their level:

```sql
WITH RECURSIVE org AS (
  SELECT id, name, 0 AS depth
  FROM employees
  WHERE manager_id IS NULL                    -- the top
  UNION ALL
  SELECT e.id, e.name, org.depth + 1
  FROM employees e
  JOIN org ON e.manager_id = org.id           -- the previous level's people are
)
SELECT name, depth FROM org ORDER BY depth, name;
```

WITH RECURSIVE, one round at a time

employees

id	name	manager_id
1	Meera	NULL
2	Arjun	1
3	Divya	1
4	Kabir	2
5	Leela	4

org (the result so far)

name	depth	
Meera	0	round 1
Arjun	1	round 2
Divya	1	
Kabir	2	round 3
Leela	3	round 4

round 5: no one → stop

1. The anchor runs once: the employee whose manager_id IS NULL. Meera is round one, at depth 0, and goes into the result.
2. The recursive member joins employees to the previous round only — Meera. Whoever reports to that person is round 2: Arjun and Divya, at depth 1.
3. The recursive member joins employees to the previous round only — Arjun and Divya. Whoever reports to them is round 3: Kabir, at depth 2.
4. The recursive member joins employees to the previous round only — Kabir. Whoever reports to that person is round 4: Leela, at depth 3.
5. The recursive member runs again with org meaning only Leela. Nobody reports to Leela, so the round is empty and recursion stops. The result is the union of all 4 rounds.

Round one is Meera. Round two joins employees whose `manager_id` is Meera's id: Arjun and Divya. Round three finds Kabir under Arjun; round four Leela under Kabir; round five finds no one under Leela and stops. The join *is* the stopping condition: leaves have no children.

To start lower — everyone under Arjun — change only the anchor: `WHERE id = 2`. To include only the subordinates and not Arjun himself, anchor on `WHERE manager_id = 2` with depth 1.

Walking up: a person's chain of command

Reverse the join to climb from one employee to the top:

```sql
WITH RECURSIVE chain AS (
  SELECT id, name, manager_id, 0 AS steps FROM employees WHERE id = 5
  UNION ALL
  SELECT m.id, m.name, m.manager_id, chain.steps + 1
  FROM employees m
  JOIN chain ON m.id = chain.manager_id
)
SELECT name, steps FROM chain ORDER BY steps;
```

This returns Leela, Kabir, Arjun, Meera — and stops when Meera's `manager_id` is NULL, since `m.id = NULL` matches nothing.

Building a path

To show where a node sits — `Electronics > Phones > Android` — carry a string column and extend it each round:

```sql
WITH RECURSIVE tree AS (
  SELECT id, name, CAST(name AS CHAR(200)) AS path
  FROM categories
  WHERE parent_id IS NULL
  UNION ALL
  SELECT c.id, c.name, CONCAT(t.path, ' > ', c.name)
  FROM categories c
  JOIN tree t ON c.parent_id = t.id
)
SELECT path FROM tree ORDER BY path;
```

The `CAST` matters in MySQL. The anchor decides each column's type: without it, `path` would be typed as wide as the longest *root* name, and a longer path built later is truncated (or rejected with "Data too long for column" in strict mode). Casting to `CHAR(200)` — or any width that fits the deepest path — avoids it. PostgreSQL has a similar rule and wants the types of both members to match exactly. Sorting by the path puts every child right after its parent, which displays a tree as an indented list.

Generating series

A recursive CTE with no table at all is a generator. Numbers:

```sql
WITH RECURSIVE seq AS (
  SELECT 1 AS n UNION ALL SELECT n + 1 FROM seq WHERE n < 100
)
SELECT n FROM seq;
```

Dates — the standard fix for reports with missing days:

```sql
WITH RECURSIVE days AS (
  SELECT '2024-03-01' AS day
  UNION ALL
  SELECT DATE_ADD(day, INTERVAL 1 DAY) FROM days WHERE day < '2024-03-07'
)
SELECT d.day, COALESCE(SUM(s.amount), 0) AS revenue
FROM days d
LEFT JOIN sales s ON s.sale_date = d.day
GROUP BY d.day
ORDER BY d.day;
```

A plain `GROUP BY sale_date` over `sales` skips a day with no sales; the date series provides every day, and the LEFT JOIN with `COALESCE` turns an empty day into 0. PostgreSQL has `generate_series()` for this; MySQL has no such function, so the recursive CTE is the idiom. Bound the series by reading the range from the data — `SELECT MIN(sale_date) …` in the anchor, `WHERE day < (SELECT MAX(sale_date) FROM sales)` in the recursive member.

Keeping recursion bounded

A recursive member that never runs out of rows would run forever. MySQL protects you with `cte_max_recursion_depth` (default 1000 rounds) and `max_execution_time`, but the query should stop on its own:

- **Counters** stop at a limit: `WHERE n < 100`.
- **Joins down a tree** stop at the leaves — but only if the data really is a tree. A cycle (A manages B, B manages A, by a data-entry mistake) loops forever.
- **Guard against cycles** by carrying a depth and stopping at a sane maximum (`WHERE org.depth < 20`), or by carrying the path of ids and refusing to revisit: `WHERE`

`FIND_IN_SET(e.id, org.id_path) = 0`. PostgreSQL 14 adds a `CYCLE` clause for this.

- **Use UNION instead of UNION ALL** when rows can be regenerated — such as walking a graph that has several routes to one node — so already-seen rows are dropped.

Recursion for aggregates

Because aggregates are not allowed inside, total-over-a-subtree questions — "how many people work under each manager, at any depth" — are answered in two steps: the recursive CTE produces one row per (manager, subordinate) pair, and the outer query groups:

```
WITH RECURSIVE reports AS (
  SELECT manager_id AS boss, id AS person FROM employees WHERE manager_id IS N
  UNION ALL
  SELECT r.boss, e.id
  FROM reports r
  JOIN employees e ON e.manager_id = r.person
)
SELECT boss, COUNT(*) AS team_size FROM reports GROUP BY boss ORDER BY boss;
```

The anchor pairs each manager with each direct report; each round extends every pair one level further down. Grouping by boss counts all of them.

In other databases

PostgreSQL and SQLite use `WITH RECURSIVE` exactly as MySQL does. SQL Server writes plain `WITH` and limits depth with `OPTION (MAXRECURSION n)` (default 100). Oracle accepts recursive `WITH` and also has its older `CONNECT BY PRIOR` syntax, which you may still meet in legacy code.

Pitfalls

- Forgetting `RECURSIVE`, or the stopping condition.
- An anchor column too narrow for the values built in later rounds.
- Cycles in data that "should" be a tree.
- Aggregating inside the recursive member instead of in the final query.
- Relying on the CTE's row order instead of an outer ORDER BY.

Key takeaways

- `WITH RECURSIVE` = anchor `UNION ALL` recursive member; each round reads only the previous round's rows; recursion stops at an empty round.
- Join on `child.parent_id = cte.id` to walk down a tree, the other way round to walk up.
- Carry a depth column for levels and a `CAST(… AS CHAR(n))` path column for readable paths.
- Generate number and date series to fill gaps before a LEFT JOIN.
- Keep recursion bounded: counters, depth limits and cycle checks; aggregate outside the CTE.

EXERCISE 1

The org chart by level

List everyone in the company with their level: 0 for anyone with no manager, 1 for their direct reports, and so on down.

employees: id	name	manager_id
1	Meera	NULL
2	Arjun	1
3	Divya	1
4	Kabir	2
5	Leela	4
6	Anil	3
7	Zara	3

Return `name` and `depth`, ordered by `depth`, then `name`.

```
name | depth
Meera | 0
Arjun | 1
Divya | 1
Anil | 2
Kabir | 2
Zara | 2
Leela | 3
```

A company may have more than one person without a manager; each starts a tree at depth 0. The depth of the chart is not fixed — use a recursive CTE.

INPUT 1

```
CREATE TABLE employees (id INT PRIMARY
KEY, name VARCHAR(30), manager_id INT);
INSERT INTO employees VALUES
  (1, 'Meera', NULL), (2, 'Arjun', 1),
(3, 'Divya', 1), (4, 'Kabir', 2),
  (5, 'Leela', 4), (6, 'Anil', 3), (7,
'Zara', 3);
```

OUTPUT 1

```
name | depth
Meera | 0
Arjun | 1
Divya | 1
Anil | 2
Kabir | 2
Zara | 2
Leela | 3
```

REFERENCE SOLUTION

```sql
WITH RECURSIVE org AS (
  SELECT id, name, 0 AS depth
  FROM employees
  WHERE manager_id IS NULL
  UNION ALL
  SELECT e.id, e.name, org.depth + 1
  FROM employees e
  JOIN org ON e.manager_id = org.id
)
SELECT name, depth FROM org ORDER BY depth, name;
```

EXERCISE 2

Full category paths

Show every category with its full path from the top, the names joined by `' > '`.

categories: id	name	parent_id
1	Electronics	NULL
2	Phones	1
3	Laptops	1
4	Android	2
5	iPhone	2
6	Books	NULL
7	Fiction	6

Return `id` and `path`, ordered by `path`.

```
id | path
6 | Books
7 | Books > Fiction
1 | Electronics
3 | Electronics > Laptops
2 | Electronics > Phones
4 | Electronics > Phones > Android
5 | Electronics > Phones > iPhone
```

In MySQL, cast the anchor's path wide enough (`CAST(name AS CHAR(200))`) so longer paths built later are not cut off.

INPUT 1

```
CREATE TABLE categories (id INT PRIMARY
KEY, name VARCHAR(40), parent_id INT);
INSERT INTO categories VALUES
  (1, 'Electronics', NULL), (2,
'Phones', 1), (3, 'Laptops', 1), (4,
'Android', 2),
  (5, 'iPhone', 2), (6, 'Books', NULL),
(7, 'Fiction', 6);
```

OUTPUT 1

```
id | path
6 | Books
7 | Books > Fiction
1 | Electronics
3 | Electronics > Laptops
2 | Electronics > Phones
4 | Electronics > Phones > Android
5 | Electronics > Phones > iPhone
```

REFERENCE SOLUTION

```sql
WITH RECURSIVE tree AS (
  SELECT id, CAST(name AS CHAR(200)) AS path
  FROM categories
  WHERE parent_id IS NULL
  UNION ALL
  SELECT c.id, CONCAT(t.path, ' > ', c.name)
  FROM categories c
  JOIN tree t ON c.parent_id = t.id
)
SELECT id, path FROM tree ORDER BY path;
```

QUIZ

1. In a recursive CTE, what does the recursive member's reference to the CTE contain on each round?

A. Every row produced so far

B. Only the rows produced by the previous round

C. Only the anchor's rows

D. The base table

2. When does `WITH RECURSIVE` stop producing rounds?

A. After 100 rounds

B. When a round produces no rows

C. When the anchor returns NULL

D. When the final ORDER BY is reached

3. A MySQL recursive CTE builds `path` starting from `name` in the anchor, and long paths come out cut short. What fixes it?

A. Use UNION instead of UNION ALL

B. Cast the anchor's column wider, e.g. `CAST(name AS CHAR(200))`

C. Add ORDER BY to the recursive member

D. Use `||` instead of CONCAT

4. Which of these is allowed in the recursive member of a MySQL recursive CTE?

A. `GROUP BY`

B. `SUM(...)`

C. A join to another table

D. `ROW_NUMBER() OVER (...)`

5. What happens if the `employees` data contains a cycle (A manages B, B manages A) and you walk it with a recursive join and `UNION ALL`?

A. The query stops at the cycle automatically

B. Rows repeat round after round until MySQL's recursion limit aborts the query

C. MySQL returns only the anchor

D. The cycle rows are skipped

ANSWER KEY · RECURSIVE CTES — HIERARCHIES, PATHS AND NUMBER SERIES

1. B Each round sees just the previous round's new rows; the final result is the union of all rounds. That is why a counter `n + 1 … WHERE n < 5` advances one step per round instead of multiplying rows.

2. B An empty round ends the recursion. MySQL also aborts a runaway query at `cte_max_recursion_depth` (1000 by default) as a safety net — the query should stop well before it, by a counter, a depth limit or running out of children.

3. B The anchor decides each column's type, so a path typed from a short root name is too narrow for longer values. `||` is logical OR in MySQL by default, and ORDER BY is not allowed in the recursive member.

4. C Joins are how recursion walks a hierarchy. Aggregates, GROUP BY, window functions, ORDER BY and DISTINCT are not allowed in the recursive member — aggregate in the outer query instead.

5. B With UNION ALL nothing notices a repeat; every round produces rows again. Guard with a depth limit or a visited-path check, or use UNION so already-produced rows are dropped.

CHECKPOINT · 25 MIN

Checkpoint — Set operations and CTEs

This checkpoint covers the whole module: stacking results with `UNION` and `UNION ALL`, the column rules and sorting the combined result, rows in both and rows in one with `INTERSECT` and `EXCEPT`, their `EXISTS` / `NOT EXISTS` and `LEFT JOIN … IS NULL` equivalents, the `NOT IN` null trap, naming steps with common table expressions, and walking hierarchies and generating series with `WITH RECURSIVE`.

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 three rules must the SELECTs of a union satisfy, and where do the result's column names come from?
- When does `UNION` give a wrong answer that `UNION ALL` would not?
- What is `A EXCEPT B` if A is `{1, 2, 3}` and B is `{2, NULL}` — and what does `NOT IN` give for the same data?
- How do you write "rows in A but not in B" on a MySQL version without `EXCEPT`?
- In a chain of CTEs, which CTEs can each one read?
- What are the anchor and the recursive member, and when does a recursive CTE stop?
- Why does a path column in a MySQL recursive CTE need a `CAST` in the anchor?

The three queries are a date-filled daily revenue report built on a recursive date series, an activity count across two tables combined with `UNION ALL` inside a CTE, and the size of every manager's whole team — direct and indirect reports — from a recursive CTE grouped in the outer query.

EXERCISE 1

Daily revenue with every day shown

The `sales` table has a row per sale. Report revenue for **every** day from the earliest sale date to the latest, including days with no sales (revenue 0).

sales: id	sale_date	amount
1	2024-03-01	100
2	2024-03-01	50
3	2024-03-03	70
4	2024-03-05	20

Return `day` and `revenue`, ordered by `day`.

```
day | revenue
2024-03-01 | 150
2024-03-02 | 0
2024-03-03 | 70
2024-03-04 | 0
2024-03-05 | 20
```

Generate the days with a recursive CTE and `DATE_ADD(day, INTERVAL 1 DAY)`; the table always has at least one sale.

INPUT 1

```
CREATE TABLE sales (id INT PRIMARY KEY,
sale_date DATE, amount INT);
INSERT INTO sales VALUES
  (1, '2024-03-01', 100), (2, '2024-03-
01', 50), (3, '2024-03-03', 70), (4,
'2024-03-05', 20);
```

OUTPUT 1

```
day | revenue
2024-03-01 | 150
2024-03-02 | 0
2024-03-03 | 70
2024-03-04 | 0
2024-03-05 | 20
```

REFERENCE SOLUTION

```sql
WITH RECURSIVE days AS (
  SELECT MIN(sale_date) AS day FROM sales
  UNION ALL
  SELECT DATE_ADD(day, INTERVAL 1 DAY)
  FROM days
  WHERE day < (SELECT MAX(sale_date) FROM sales)
)
SELECT d.day, COALESCE(SUM(s.amount), 0) AS revenue
FROM days d
LEFT JOIN sales s ON s.sale_date = d.day
GROUP BY d.day
ORDER BY d.day;
```

EXERCISE 2

Activity across posts and comments

Count each user's activity: the number of their posts plus the number of their comments. Users with no activity show 0.

users: id	name
1	Asha
2	Ben
3	Chen
4	Dev

posts: id	user_id
1	1
2	1
3	2

comments: id	user_id
1	2
2	2
3	1
4	3

Return `name` and `activity`, ordered by `activity` descending, then `name`.

```
name | activity
Asha | 3
Ben | 3
Chen | 1
Dev | 0
```

INPUT 1

```
CREATE TABLE users (id INT PRIMARY KEY,
name VARCHAR(30));
CREATE TABLE posts (id INT PRIMARY KEY,
user_id INT);
CREATE TABLE comments (id INT PRIMARY
KEY, user_id INT);
INSERT INTO users VALUES (1, 'Asha'),
(2, 'Ben'), (3, 'Chen'), (4, 'Dev');
INSERT INTO posts VALUES (1, 1), (2, 1),
(3, 2);
INSERT INTO comments VALUES (1, 2), (2,
2), (3, 1), (4, 3);
```

OUTPUT 1

```
name | activity
Asha | 3
Ben | 3
Chen | 1
Dev | 0
```

REFERENCE SOLUTION

```sql
WITH actions AS (
  SELECT user_id FROM posts
  UNION ALL
  SELECT user_id FROM comments
)
SELECT u.name, COUNT(a.user_id) AS activity
FROM users u
LEFT JOIN actions a ON a.user_id = u.id
GROUP BY u.id, u.name
ORDER BY activity DESC, u.name;
```

EXERCISE 3

Every manager's whole team

For each employee who manages anyone, count everyone below them — direct reports, their reports, and so on.

employees: id	name	manager_id
1	Meera	NULL
2	Arjun	1
3	Divya	1
4	Kabir	2
5	Leela	4
6	Anil	3

Return `name` and `team_size`, ordered by `team_size` descending, then `name`. People who manage no one do not appear.

```
name | team_size
Meera | 5
Arjun | 2
Divya | 1
Kabir | 1
```

INPUT 1

```
CREATE TABLE employees (id INT PRIMARY
KEY, name VARCHAR(30), manager_id INT);
INSERT INTO employees VALUES
  (1, 'Meera', NULL), (2, 'Arjun', 1),
(3, 'Divya', 1), (4, 'Kabir', 2), (5,
'Leela', 4), (6, 'Anil', 3);
```

OUTPUT 1

```
name | team_size
Meera | 5
Arjun | 2
Divya | 1
Kabir | 1
```

REFERENCE SOLUTION

```sql
WITH RECURSIVE reports AS (
  SELECT manager_id AS boss, id AS person
  FROM employees
  WHERE manager_id IS NOT NULL
  UNION ALL
  SELECT r.boss, e.id
  FROM reports r
  JOIN employees e ON e.manager_id = r.person
)
SELECT m.name, COUNT(*) AS team_size
FROM reports r
JOIN employees m ON m.id = r.boss
GROUP BY r.boss, m.name
ORDER BY team_size DESC, m.name;
```

CHECKPOINT QUESTIONS

1. Which statement is refused?

A. `SELECT a, b FROM t1 UNION SELECT c, d FROM t2`

B. `SELECT a, b FROM t1 UNION SELECT c FROM t2`

C. `SELECT a FROM t1 UNION ALL SELECT c FROM t2 ORDER BY a`

D. `SELECT a FROM t1 UNION SELECT 'x'`

2. A union's branches are tagged with `'post'` and `'comment'` in a `kind` column. What changes if you switch `UNION ALL` to `UNION`?

A. Nothing is removed between the branches, but identical rows *within* one branch are now collapsed

B. Nothing at all, ever

C. The kind column is dropped

D. The result is sorted by kind

3. `SELECT x FROM a EXCEPT SELECT x FROM b` — `a.x` is {1, 1, 2, 3}, `b.x` is {3}. What is returned?

A. 1, 1, 2

B. 1, 2

C. 2

D. 3

4. Which way of writing "customers with no orders" returns no rows at all if `orders.customer_id` contains a NULL?

A. `WHERE NOT EXISTS (SELECT 1 FROM orders o WHERE o.customer_id = c.id)`

B. `LEFT JOIN orders o ON o.customer_id = c.id WHERE o.customer_id IS NULL`

C. `WHERE c.id NOT IN (SELECT customer_id FROM orders)`

D. `SELECT id FROM customers EXCEPT SELECT customer_id FROM orders`

5. Which is the correct way to emulate `A INTERSECT B` on a single `id` column in MySQL 5.7?

A. `SELECT DISTINCT id FROM A WHERE id IN (SELECT id FROM B)`

B. `SELECT id FROM A UNION SELECT id FROM B`

C. `SELECT id FROM A WHERE id NOT IN (SELECT id FROM B)`

D. `SELECT id FROM A CROSS JOIN B`

6. In `WITH a AS (…), b AS (…), c AS (…) SELECT …`, which CTEs can `b` read?

A. `a` only

B. `a` and `c`

C. `c` only

D. None — CTEs cannot read each other

7. Why would you choose a temporary table over a CTE?

A. A CTE cannot contain joins

B. The intermediate result must be read by several separate statements, or needs its own index

C. Temporary tables are always faster

D. CTEs cannot be used with GROUP BY

8. What does this return?

```sql
WITH RECURSIVE s AS (
  SELECT 2 AS n
  UNION ALL
  SELECT n * 2 FROM s WHERE n < 20
)
SELECT COUNT(*) FROM s;
```

A. 4

B. 5

C. 10

D. It never stops

9. In the anchor of an org-chart recursive CTE, which condition selects the top of the hierarchy?

A. `WHERE manager_id = 0`

B. `WHERE manager_id IS NULL`

C. `WHERE manager_id = NULL`

D. `WHERE id = manager_id`

10. To fill missing days in a daily report, you generate dates with a recursive CTE. How do you join the data?

A. `INNER JOIN sales` on the date

B. `LEFT JOIN sales` from the dates, then `COALESCE(SUM(amount), 0)`

C. `RIGHT JOIN` the dates to sales

D. `UNION ALL` the dates with sales

11. Which operation binds tightest in MySQL and standard SQL when mixed without parentheses?

A. UNION

B. EXCEPT

C. INTERSECT

D. They all bind equally, left to right

12. Where does the aggregation go when you want the total size of each manager's subtree?

A. Inside the recursive member, with GROUP BY boss

B. In the anchor

C. In the outer query that reads the recursive CTE

D. It cannot be done in SQL

ANSWER KEY · CHECKPOINT — SET OPERATIONS AND CTES

1. B Every SELECT of a union needs the same number of columns. Different names are fine (the first SELECT names the result), a trailing ORDER BY sorts the whole result, and a SELECT with no FROM is a valid branch.

2. A The tag makes rows of different branches different, so nothing is removed *between* branches — but two identical rows inside one branch are still collapsed by UNION. UNION does not promise any order.

3. B EXCEPT returns the *distinct* rows of the first result not found in the second: 1 and 2. (`EXCEPT ALL`, where supported, would keep both 1s.)

4. C NOT IN compares with `<>` against every value, and `<> NULL` is unknown, so no row passes. The other three are unaffected by a NULL in `orders.customer_id`.

5. A A semi-join (IN or EXISTS) keeps A's rows that have a match in B; DISTINCT matches INTERSECT's duplicate removal. UNION is "either", NOT IN is the difference, and a cross join pairs every row with every row.

6. A A CTE can read the CTEs defined before it in the same WITH list. The final SELECT can read all three.

7. B A CTE lives for one statement and has no indexes of its own; a temporary table lasts for the session, can be indexed and read many times. CTEs can join and group freely.

8. B Rounds: 2, 4, 8, 16, 32. From 32 the condition `n < 20` fails, so that round is empty. 32 itself was produced from 16 (which passed), so five rows.

9. B The person with no manager has `manager_id` NULL, and NULL is tested with `IS NULL` — `= NULL` is never true.

10. B The dates must all survive, so they are the preserved side of a LEFT JOIN; a day with no sales then has NULL amounts, and `SUM` of nothing is NULL until COALESCE makes it 0. An inner join would drop the empty days again.

11. C INTERSECT binds tighter than UNION and EXCEPT, so `A UNION B INTERSECT C` is `A UNION (B INTERSECT C)`. SQLite is the exception, evaluating left to right — parentheses or CTEs avoid the question.

12. C The recursive member may not aggregate. Let the CTE produce one row per (boss, person) pair at every depth and group them outside.
