Module 3 of 16

Sorting and limiting

ORDER BY with ASC and DESC, aliases, expressions and where NULLs sort; several keys, tie-breakers and custom orders; LIMIT and OFFSET with MySQL's comma form, TOP and FETCH FIRST; offset and keyset pagination; top-N queries and the Nth highest value.

A table has no order, so a query has none either until it asks for one. This module teaches the clauses that turn a set of rows into a list you can read, page through and rank: `ORDER BY` to sort, `LIMIT` and `OFFSET` to take a slice. They are short clauses with long consequences — almost every report, feed, leaderboard and API page is built on them, and the classic interview questions ("second highest salary", "why do rows repeat across pages", "where do NULLs sort") live here. The examples are MySQL 8, with the places where PostgreSQL, SQL Server and Oracle differ called out, because interviews ask about those too.

ORDER BY covers ascending and descending keys, sorting by aliases, expressions and positions, why aliases work there but not in `WHERE`, where MySQL puts NULLs and the `col IS NULL` idiom, and how collation changes string order. Sorting on several keys covers left-to-right comparison, per-key directions, unique tie-breakers, boolean keys and custom orders with `CASE` and `FIELD`. LIMIT and OFFSET covers the slice, the comma form, the processing order and the other dialects' `TOP` and `FETCH FIRST`. Pagination covers the offset formula, the duplicate-row bug, deep-offset cost and keyset pagination. Top-N queries covers top and bottom N, the Nth highest value with `DISTINCT`, returning NULL with a scalar subquery, ties at the cut-off and why per-group top-N needs more.

The exercises are queries run against small tables: a price list with a tie-breaker, hire dates with unknowns last, a class ranking on three keys, a support queue in priority order, the latest orders, a middle slice, page three of a catalogue, the page after a keyset cursor, the cheapest items in stock and the second highest salary. The checkpoint adds a leaderboard, a catalogue page with unpriced items last and the third highest distinct score.

CONTENTS

1	ORDER BY — putting rows in order	14 min
2	Sorting on several keys, ties and custom orders	14 min
3	LIMIT and OFFSET — taking a slice of the result	13 min

4	Pagination — offset pages and keyset pages	15 min
5	Top-N queries and the Nth highest value	15 min
✓	Checkpoint — Sorting and limiting	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

ORDER BY — putting rows in order

A table is a set of rows, and a set has no order. When you write `SELECT name FROM products`, the database hands back the rows in whatever order is cheapest for it — often the order they were inserted, sometimes the order of an index it happened to use, sometimes something else after the table has been updated a few times. That order is an accident of storage, not a promise. The moment the order of the rows matters to you — a price list, a leaderboard, the newest orders first, or an exercise judged line by line — the query needs an `ORDER BY` clause. This lesson covers how it works, ascending and descending keys, sorting by aliases, expressions and positions, where NULLs land, and how strings compare.

The clause and its place

`ORDER BY` comes after `WHERE` (and, later in the course, after `GROUP BY` and `HAVING`), and before `LIMIT`:

```
SELECT name, price
FROM products
WHERE in_stock = 1
ORDER BY price;
```

name	price
pencil	5
notebook	40
backpack	900

The keyword is followed by a **sort key**. Rows are compared on that key and put in ascending order: smallest number first, earliest date first, alphabetically first string first. `ASC` says that explicitly; it is the default, so `ORDER BY price` and `ORDER BY price ASC` are the same query.

Remember the logical order in which a query is processed: `FROM` → `WHERE` → `SELECT` → `ORDER BY` → `LIMIT`. Sorting happens **after** the select list has been computed and **after** filtering. Two consequences follow. First, `ORDER BY` only ever sorts the rows that survived

`WHERE` — it cannot bring filtered rows back. Second, because the select list already exists when sorting starts, `ORDER BY` can refer to its aliases.

Descending order

Write `DESC` after a key to reverse it:

```
SELECT name, price
FROM products
ORDER BY price DESC;
```

ORDER BY moves rows into place; without it there is no order

ORDER BY discount IS NULL, discount

products

	name	price	discount	
1	backpack	900	5	discount IS NULL = 0
2	notebook	40	10	discount IS NULL = 0
3	pencil	5	NULL	discount IS NULL = 1

1. Without ORDER BY the rows come back in whatever order is cheapest for the database — here, say, the order they happen to be stored in. That order is not a promise.
2. ORDER BY price sorts on the key, ascending by default: pencil 5, notebook 40, backpack 900.
3. DESC after a key reverses that key: the most expensive first. DESC belongs to the one key it follows, not to the whole clause.
4. Sorting on discount, MySQL treats NULL as smaller than every value, so the product with no discount comes first ascending (and would come last descending).
5. To send NULLs to the end, sort first on the hidden key discount IS NULL — 0 for a value, 1 for NULL — and then on discount within each group.

`DESC` belongs to the one key it follows, not to the whole clause — that matters as soon as there are several keys, which is the next lesson. A common slip is writing `ORDER BY DESC price`; the direction always comes after the key.

Sorting by an alias or an expression

Any expression can be a sort key, and so can a name from the select list:

```sql
SELECT name, price, qty, price * qty AS stock_value
FROM products
ORDER BY stock_value DESC;
```

That works because the alias `stock_value` exists by the time the sort runs. The same alias in `WHERE stock_value > 100` fails, because `WHERE` runs before `SELECT` — a classic interview question about the logical processing order. You can also sort by the expression itself, written out again, even if it is not selected:

```sql
SELECT name
FROM products
ORDER BY price * qty DESC;
```

Sorting on a column you do not display is perfectly legal in an ordinary query: `SELECT name FROM employees ORDER BY hire_date` lists names oldest-hire first. The one exception is `SELECT DISTINCT`: there the sort key must be in the select list, because after duplicates are removed one output row may stand for several input rows with different `hire_date` values, and the database cannot tell which one to sort by. MySQL refuses such a query with an error mentioning the select list; PostgreSQL does too.

Sorting by position

`ORDER BY 2` means "by the second column of the select list":

```sql
SELECT name, price FROM products ORDER BY 2 DESC;
```

It is short, and you will see it in quick ad-hoc queries and in some interview answers. It is also fragile: add a column to the front of the select list and the query silently sorts by something else. In code that lives longer than a minute, name the column or alias instead.

Where NULLs go

A `NULL` is "no value", so where should it sort? The standard leaves it to the vendor, and the vendors disagree — which is exactly why interviewers ask.

Database	`ASC`	`DESC`
MySQL, SQLite, SQL Server	NULLs first	NULLs last
PostgreSQL, Oracle	NULLs last	NULLs first

MySQL behaves as if NULL were smaller than every value:

```
SELECT name, discount FROM products ORDER BY discount;
```

name	discount
pencil	NULL
backpack	5
notebook	10

PostgreSQL and Oracle let you say what you want with `NULLS FIRST` / `NULLS LAST`. MySQL has no such syntax, so the MySQL idiom is to sort on a second, hidden key first: the expression `discount IS NULL`, which is `0` for a real value and `1` for a NULL.

```
-- MySQL: real discounts ascending, NULLs at the end
SELECT name, discount
FROM products
ORDER BY discount IS NULL, discount;
```

name	discount
backpack	5
notebook	10
pencil	NULL

The rows are sorted by the first key (`0` before `1`, so values before NULLs), and within each group by `discount`. That pattern — prefix a boolean key to move a class of rows to one end — comes back in the next lesson for custom orders.

How strings and dates compare

Numbers compare as numbers. Strings compare character by character according to the column's **collation** — the rules for which characters sort before which. MySQL 8's default collation (`utf8mb4_0900_ai_ci`) is case-insensitive and accent-insensitive, so `'apple'`, `'Apple'` and `'APPLE'` sort as equals. SQLite and PostgreSQL's common defaults compare by code point, where every uppercase letter comes before every lowercase one, so `'Zebra'` sorts before `'apple'`. If an interview asks why the same `ORDER BY name` returns a different order on two databases, collation is the answer.

Two traps follow from comparing strings as strings. A number stored in a text column sorts as text: `'10'` comes before `'9'`, because `'1'` is less than `'9'`. And a date stored as text sorts correctly only in `'YYYY-MM-DD'` form, where text order and time order agree — `'03/01/2024'` style dates do not. Real `DATE` columns, the ones you should use, sort by time.

Ties and determinism

When two rows have the same sort key, `ORDER BY` says nothing about which comes first. The database may return them in either order, and the order can change between runs. If a printed result must be exactly reproducible — and every judged exercise in this course is — break the tie with another key, ideally one that is unique such as the primary key:

```sql
SELECT name, price
FROM products
ORDER BY price DESC, id;
```

The next lesson is about exactly that: sorting on several keys.

Before you move on

- Without `ORDER BY`, row order is undefined; never rely on insertion order.
- `ASC` is the default; `DESC` applies to the one key it follows.
- `ORDER BY` runs after `SELECT`, so aliases work there but not in `WHERE`.
- MySQL puts NULLs first ascending and last descending; `ORDER BY col IS NULL, col` moves them to the end.
- Collation decides how strings compare, and text numbers sort as text.

EXERCISE 1

Price list, most expensive first

The `products` table:

id	name	price
1	notebook	40
2	pencil	5
3	backpack	900
4	eraser	5
5	stapler	120

List every product as `name`, `price`, the most expensive first. Products with the same price are listed alphabetically by name.

```
name | price
backpack | 900
stapler | 120
notebook | 40
eraser | 5
pencil | 5
```

INPUT 1

```
CREATE TABLE products (id INT PRIMARY
KEY, name VARCHAR(40), price INT);
INSERT INTO products VALUES (1,
'notebook', 40), (2, 'pencil', 5), (3,
'backpack', 900), (4, 'eraser', 5), (5,
'stapler', 120);
```

OUTPUT 1

```
name | price
backpack | 900
stapler | 120
notebook | 40
eraser | 5
pencil | 5
```

REFERENCE SOLUTION

```
SELECT name, price
FROM products
ORDER BY price DESC, name;
```

EXERCISE 2

Hire dates, unknowns last

The `employees` table; `hired_on` is NULL when the date was never recorded:

id	name	hired_on
1	Asha	2021-03-15
2	Ben	NULL
3	Chen	2019-07-01
4	Dina	2021-03-15
5	Eli	NULL

List `name`, `hired_on` from the earliest hire to the latest, with the NULL dates **at the end**. Rows with the same date (or both NULL) are ordered by `id`.

```
name | hired_on
Chen | 2019-07-01
Asha | 2021-03-15
Dina | 2021-03-15
Ben | NULL
Eli | NULL
```

INPUT 1

```
CREATE TABLE employees (id INT PRIMARY
KEY, name VARCHAR(40), hired_on DATE);
INSERT INTO employees VALUES (1, 'Asha',
'2021-03-15'), (2, 'Ben', NULL), (3,
'Chen', '2019-07-01'), (4, 'Dina',
'2021-03-15'), (5, 'Eli', NULL);
```

OUTPUT 1

```
name | hired_on
Chen | 2019-07-01
Asha | 2021-03-15
Dina | 2021-03-15
Ben | NULL
Eli | NULL
```

REFERENCE SOLUTION

```sql
SELECT name, hired_on
FROM employees
ORDER BY hired_on IS NULL, hired_on, id;
```

QUIZ

1. What order does `SELECT name FROM products` (no ORDER BY) return its rows in?

A. Insertion order

B. Primary key order

C. Alphabetical order

D. No guaranteed order

2. Why does this work in MySQL?

```
SELECT price * qty AS total FROM items ORDER BY total DESC;
```

while `WHERE total > 100` on the same query fails?

A. ORDER BY is evaluated after SELECT, WHERE before it

B. Aliases only work with DESC

C. MySQL allows aliases everywhere except WHERE by special exception

D. WHERE needs the alias in quotes

3. In MySQL, `SELECT name, discount FROM products ORDER BY discount DESC` — where do rows with a NULL discount appear?

A. First

B. Last

C. Excluded from the result

D. In insertion order among the others

4. A `code` column is `VARCHAR` and holds `'9'`, `'10'` and `'100'`. What does `ORDER BY code` give?

A. 9, 10, 100

B. 10, 100, 9

C. 100, 10, 9

D. An error

5. What does `ORDER BY 2` mean in `SELECT name, price FROM products ORDER BY 2`?

A. Sort by the constant 2 (no effect)

B. Sort by the second column of the select list, price

C. Return only 2 rows

D. Sort by the table's second column

ANSWER KEY · ORDER BY — PUTTING ROWS IN ORDER

1. D A table is a set; without `ORDER BY` the engine returns rows in whatever order its plan produces. It often *looks* like insertion or key order, which is why the bug survives until an index or an update changes the plan.

2. A The logical order is FROM → WHERE → SELECT → ORDER BY → LIMIT. The alias is created in SELECT, so it exists by the time ORDER BY runs but not yet in WHERE. Quoting it in WHERE would compare against a string literal, not the column.

3. B MySQL treats NULL as lower than any value: first when ascending, last when descending. ORDER BY never removes rows. PostgreSQL would put them first here, which is the dialect difference interviews probe.

4. B Text compares character by character: `'1'` < `'9'`, so both strings starting with 1 come before `'9'`, and `'10'` is a prefix of `'100'`. Numbers stored as text sort as text; cast them (next module) or store them as numbers.

5. B A bare integer in ORDER BY is a position in the select list. It is legal but fragile: reordering the select list silently changes the sort. LIMIT, not ORDER BY, restricts the number of rows.

LESSON 2 · 14 MIN

Sorting on several keys, ties and custom orders

Real orderings almost always have more than one rule. A class list is sorted by class, then within each class by score, best first, then by name when two students share a score. A support queue shows urgent tickets first, then the oldest. A leaderboard ranks by points, then by who reached them first. SQL expresses each of those with one `ORDER BY` clause holding several **sort keys**, each with its own direction. This lesson covers how the keys combine, mixing ascending and descending, why the last key should be unique, and how to sort by an order that is not alphabetical or numeric — the custom order.

Keys are compared left to right

```
SELECT class, name, score
FROM students
ORDER BY class, score DESC, name;
```

The database compares two rows on the first key, `class`. If they differ, that decides their order and the other keys are never looked at. Only when the classes are equal does it compare `score` (descending), and only if those are equal too does it compare `name`.

class	name	score
A	Meera	91
A	Arjun	85
A	Tara	85
B	Kiran	78
B	Dev	60

Arjun and Tara share a class and a score, so the third key, `name`, puts Arjun first. Kiran's 78 sorts below Meera's 91 not because of the score but because class `B` comes after class `A`: a later key never overrides an earlier one. That is what people mean by "sort by class, then by score": a dictionary-style, lexicographic comparison.

Each key has its own direction

`ASC` and `DESC` attach to the key they follow. In `ORDER BY class, score DESC, name` the class and the name are ascending and only the score is descending. A very common bug is writing `ORDER BY score, name DESC` while meaning "both descending" — that sorts scores from lowest to highest. To reverse both:

```
ORDER BY score DESC, name DESC
```

Mixing directions is normal: "newest first, then alphabetical" is `ORDER BY created_at DESC, title`.

Make the last key unique

Two rows that tie on every key may come back in either order. The database is free to choose, and the choice can change from one run to the next — when the table grows, when the plan switches to an index, when a parallel scan finishes in a different order. SQL also makes no promise that sorting is **stable**: you cannot sort by name in one step and rely on that order surviving a later sort by score.

So if the order matters, finish with a key that no two rows share, usually the primary key:

```
SELECT id, name, points
FROM players
ORDER BY points DESC, id;
```

Now every row has exactly one place. This is not pedantry. Pagination (two lessons ahead) skips and repeats rows when the order is not unique, test output flips between runs, and a "first come, first served" ranking is only fair if the tie-breaker says who came first — a `joined_at` timestamp, then the id in case two players joined in the same second.

Sorting on computed keys

A key can be any expression, which lets you sort on facts the table does not store directly:

```sql
-- Longest titles first, then alphabetical
SELECT title
FROM books
ORDER BY CHAR_LENGTH(title) DESC, title;

-- Biggest discount in money, not in percent
SELECT name, price, discount_pct
FROM products
ORDER BY price * discount_pct / 100 DESC, id;
```

Boolean expressions are especially useful. In MySQL a comparison yields `1` for true and `0` for false, so sorting on a condition moves the matching rows to one end:

```sql
-- Featured products first (featured = 'yes' gives 1), then by price
SELECT name, featured, price
FROM products
ORDER BY featured = 'yes' DESC, price;
```

You saw the same trick with `ORDER BY col IS NULL, col` for NULLs-last in the previous lesson.

Custom orders with CASE

Statuses, priorities, sizes and weekdays have a natural order that is neither alphabetical nor numeric. Alphabetically, `high` < `low` < `medium`, which no one wants. The portable fix is to map each value to a rank with a `CASE` expression and sort on the rank:

```sql
SELECT id, title, priority
FROM tickets
ORDER BY CASE priority
           WHEN 'high'   THEN 1
           WHEN 'medium' THEN 2
           WHEN 'low'    THEN 3
           ELSE 4
         END,
         id;
```

id	title	priority
2	Checkout broken	high
5	Refund stuck	high
1	Typo on FAQ	medium
4	Dark mode	low

`CASE` gets a full lesson in the next module; for now read it as "if `priority` is `'high'` the key is 1, if `'medium'` it is 2", and so on. The `ELSE 4` sends unexpected values, including NULL, to the end instead of the front — without it they would get NULL, which MySQL sorts first.

MySQL has a shorthand: `FIELD(value, 'a', 'b', 'c')` returns the position of `value` in the list (1, 2, 3) and `0` when it is not there.

```
ORDER BY FIELD(priority, 'high', 'medium', 'low'), id
```

Beware the `0`: an unlisted value sorts **before** `high`. If the data can hold other values, use `CASE` with an `ELSE`, or list the order backwards and sort `DESC`. `FIELD` is MySQL-only; `CASE` works everywhere, which is why interview answers usually use it. A third option, when the order is real data rather than presentation, is to store a rank column (or a lookup table with one) and sort on that.

Sorting rows of a few groups

A multi-key sort is also how you make a report readable before you know `GROUP BY`. Sorting by department, then by salary descending, lays out each department as a block with its best-paid people at the top:

```
SELECT dept, name, salary
FROM employees
ORDER BY dept, salary DESC, name;
```

Picking just the top row of each block is a different problem — "top-N per group" — and needs window functions, which come later in the course. Sorting only arranges rows; it never removes them.

Before you move on

- Keys compare left to right; a later key only breaks ties left by the earlier ones.
- `ASC` / `DESC` belong to one key each.
- SQL sorting is not stable; end with a unique key when order matters.
- Expressions and boolean conditions make good keys.
- Custom orders: `CASE` with an `ELSE`, or MySQL's `FIELD` (unlisted values give 0 and sort first).

EXERCISE 1

Class ranking

The `students` table:

id	name	class	score
1	Tara	A	85
2	Dev	B	60
3	Meera	A	91
4	Kiran	B	78
5	Arjun	A	85

List `class`, `name`, `score`: classes in alphabetical order, within a class the highest score first, and students with the same class and score alphabetically by name.

```
class | name | score
A | Meera | 91
A | Arjun | 85
A | Tara | 85
B | Kiran | 78
B | Dev | 60
```

INPUT 1

```
CREATE TABLE students (id INT PRIMARY
KEY, name VARCHAR(40), class CHAR(1),
score INT);
INSERT INTO students VALUES (1, 'Tara',
'A', 85), (2, 'Dev', 'B', 60), (3,
'Meera', 'A', 91), (4, 'Kiran', 'B',
78), (5, 'Arjun', 'A', 85);
```

OUTPUT 1

```
class | name | score
A | Meera | 91
A | Arjun | 85
A | Tara | 85
B | Kiran | 78
B | Dev | 60
```

REFERENCE SOLUTION

```sql
SELECT class, name, score
FROM students
ORDER BY class, score DESC, name;
```

EXERCISE 2

Support queue in priority order

The `tickets` table:

id	title	priority	opened_on
1	Typo on FAQ	medium	2024-05-03
2	Checkout broken	high	2024-05-04
3	Add dark mode	low	2024-05-01
4	Login loop	high	2024-05-02
5	Slow search	medium	2024-05-03
6	Old ticket	urgent	2024-04-01

List `id`, `title`, `priority` in queue order: `high` first, then `medium`, then `low`, then **any other value, including NULL**, last. Within a priority, the oldest `opened_on` first; then by `id`.

```
id | title | priority
4 | Login loop | high
2 | Checkout broken | high
1 | Typo on FAQ | medium
5 | Slow search | medium
3 | Add dark mode | low
6 | Old ticket | urgent
```

INPUT 1

```
CREATE TABLE tickets (id INT PRIMARY
KEY, title VARCHAR(60), priority
VARCHAR(10), opened_on DATE);
INSERT INTO tickets VALUES
  (1, 'Typo on FAQ', 'medium', '2024-05-
03'),
  (2, 'Checkout broken', 'high', '2024-
05-04'),
  (3, 'Add dark mode', 'low', '2024-05-
01'),
  (4, 'Login loop', 'high', '2024-05-
02'),
  (5, 'Slow search', 'medium', '2024-05-
03'),
  (6, 'Old ticket', 'urgent', '2024-04-
01');
```

OUTPUT 1

```
id | title | priority
4 | Login loop | high
2 | Checkout broken | high
1 | Typo on FAQ | medium
5 | Slow search | medium
3 | Add dark mode | low
6 | Old ticket | urgent
```

REFERENCE SOLUTION

```sql
SELECT id, title, priority
FROM tickets
ORDER BY CASE priority
           WHEN 'high' THEN 1
           WHEN 'medium' THEN 2
           WHEN 'low' THEN 3
           ELSE 4
         END,
         opened_on,
         id;
```

QUIZ

1. Which rows are compared on `name` in `ORDER BY dept, salary DESC, name`?

A. Every row

B. Only rows with the same dept

C. Only rows with the same dept and the same salary

D. Only rows with the same salary

2. What does `ORDER BY score, name DESC` do?

A. Both keys descending

B. Score ascending, name descending

C. Score descending, name ascending

D. It is a syntax error

3. `ORDER BY FIELD(size, 'S', 'M', 'L')` runs on rows whose sizes are `M`, `XL` and `S`. What order do they come out in?

A. S, M, XL

B. XL, S, M

C. S, M, then XL anywhere

D. An error, because XL is not listed

4. Why end an ORDER BY with the primary key?

A. It makes the query faster

B. It gives every row exactly one position, so the order is reproducible

C. MySQL requires it when there are several keys

D. It removes duplicate rows

5. In MySQL, `SELECT name FROM products ORDER BY featured = 'yes' DESC, price` — what does the first key do?

A. Filters to featured products

B. Puts products with `featured = 'yes'` first, since the comparison is 1 for them and 0 for others

C. Is a syntax error; ORDER BY takes only columns

D. Sorts the featured column alphabetically

ANSWER KEY · SORTING ON SEVERAL KEYS, TIES AND CUSTOM ORDERS

1. C Keys are compared left to right, and a later key only decides between rows that tie on all earlier keys. Two rows with the same salary but different departments are already ordered by dept.

2. B `DESC` attaches to the one key it follows. To sort both descending write `ORDER BY score DESC, name DESC`.

3. B `FIELD` returns the 1-based position in the list and `0` for a value that is not listed, so `XL` gets 0 and sorts first. A `CASE` with an `ELSE` rank is the safe way to send unknown values to the end.

4. B Rows tied on all keys come back in an undefined order, and SQL sorting is not stable. A unique last key removes every tie. It is not required by syntax and has nothing to do with DISTINCT.

5. B A comparison is an expression yielding 1 or 0 in MySQL, so it can be a sort key; `DESC` puts the 1s first. ORDER BY never filters rows — WHERE does.

LESSON 3 · 13 MIN

LIMIT and OFFSET — taking a slice of the result

Most of the time you do not want every row a query can produce. A home page shows the five newest posts, a dashboard the ten biggest customers, an API the first page of results, and an interview question asks for "the highest salary". `LIMIT` cuts a result down to its first rows, and `OFFSET` skips some rows before the cut starts. They are small clauses, but they come with a rule people forget — they mean nothing without `ORDER BY` — and with more dialect differences than almost anything else in SQL. This lesson covers both clauses, MySQL's comma form, where they sit in the processing order, and the `TOP` and `FETCH FIRST` spellings you will meet in other databases.

LIMIT

```sql
SELECT id, customer, total
FROM orders
ORDER BY total DESC
LIMIT 3;
```

id	customer	total
14	Ravi	4200
3	Anita	3900
9	Sam	2750

`LIMIT 3` keeps at most three rows. "At most": if the query only produces two rows, you get two; if it produces none, you get none. `LIMIT 0` is legal and returns no rows at all — tools sometimes use it to learn a query's column names without fetching data.

`LIMIT` is the very last step of the query. In the logical order — `FROM`, `WHERE`, `SELECT`, `ORDER BY`, `LIMIT` — the rows are filtered, computed and sorted first, and only then is the first slice taken. So `WHERE` decides which rows are candidates, `ORDER BY` decides which of them are "first", and `LIMIT` decides how many of those you keep.

Always sort before you limit

`LIMIT` takes the first rows of the result's order. Without `ORDER BY` there is no defined order, so:

```sql
SELECT id, customer FROM orders LIMIT 3;   -- some three orders
```

returns *some* three orders. Today they may be the first three inserted; after a delete, an update or a new index, they may be three others. This is the single most common `LIMIT` bug in code review and the reason an interviewer frowns at a "top" query without a sort. And because ties are unordered too, a limited result is only reproducible when the sort is unique: `ORDER BY total DESC, id`. With two orders tied at the cut-off, `ORDER BY total DESC LIMIT 3` may return either one.

OFFSET

`OFFSET m` skips the first `m` rows of the sorted result, then `LIMIT` counts from there:

```sql
SELECT id, customer, total
FROM orders
ORDER BY total DESC, id
LIMIT 3 OFFSET 3;      -- rows 4, 5 and 6
```

LIMIT keeps a slice of the sorted rows; OFFSET says where it starts

… LIMIT 3 OFFSET 9

orders

	id	customer	total
1	14	Ravi	4200
2	3	Anita	3900
3	9	Sam	2750
4	5	Meera	2100
5	11	Kiran	1800
6	2	Dev	1500
7	7	Tara	900
8	12	Omar	650

no rows — an empty result, not an error

1. The 8 orders as storage hands them back. A LIMIT on this would take some three rows — whichever came first today.
2. ORDER BY total DESC, id runs before LIMIT and decides which rows are first. id breaks any tie, so the order is the same on every run.
3. LIMIT 3 is the very last step: it keeps the first three sorted rows — the top three orders, led by Ravi at 4200.
4. OFFSET 3 skips three rows first, counting from zero, and LIMIT counts from there: rows 4 to 6, the second page.
5. OFFSET 6 leaves only 2 rows. LIMIT means at most, so the third page is short rather than padded or an error.
6. An offset past the end, OFFSET 9 on 8 rows, skips everything: the result is the column headings and no rows.

Offsets count from zero: `OFFSET 0` skips nothing, `OFFSET 3` starts at the fourth row. An offset past the end is not an error — the result is simply empty. In MySQL an `OFFSET` must come with a `LIMIT`; to skip rows and keep "all the rest", the documented trick is a huge limit such as `LIMIT 18446744073709551615 OFFSET 10`. That need is rare.

MySQL's comma form

MySQL also accepts two numbers separated by a comma:

```
LIMIT 3, 5     -- skip 3, then take 5   (the same as LIMIT 5 OFFSET 3)
```

The comma form puts the **offset first** and the count second, the opposite of the reading order of `LIMIT 5 OFFSET 3`. It is everywhere in older PHP tutorials and an easy point for a trick question: `LIMIT 10, 20` returns twenty rows, starting with the eleventh. SQLite accepts it

with the same meaning; PostgreSQL does not. Prefer `LIMIT … OFFSET …` in anything you write — the words say which number is which.

One more MySQL detail: the numbers must be literal non-negative integers or `?` placeholders in a prepared statement. `LIMIT 2 + 1` and `LIMIT (SELECT …)` are syntax errors in MySQL, though some other engines accept expressions there.

The other spellings

`LIMIT` is not in the SQL standard, and every database family has its own way to say it. You will see all of them in interviews:

Database	First 5 rows	Rows 11–15
MySQL, SQLite	`LIMIT 5`	`LIMIT 5 OFFSET 10`
PostgreSQL	`LIMIT 5` or `FETCH FIRST 5 ROWS ONLY`	`OFFSET 10 LIMIT 5`
SQL Server	`SELECT TOP (5) …`	`OFFSET 10 ROWS FETCH NEXT 5 ROWS ONLY`
Oracle 12c+	`FETCH FIRST 5 ROWS ONLY`	`OFFSET 10 ROWS FETCH NEXT 5 ROWS ONLY`
Older Oracle	`WHERE ROWNUM ⩽ 5` (in a subquery)	nested `ROWNUM` subqueries

The standard form, `OFFSET … ROWS FETCH FIRST|NEXT … ROWS ONLY`, also offers `FETCH FIRST 3 ROWS WITH TIES`, which keeps any extra rows tied with the last one — PostgreSQL 13+, Oracle and SQL Server's `TOP (3) WITH TIES` support it, MySQL does not. In MySQL, "top three including ties" needs a subquery or a window function, both later in the course. The Oracle `ROWNUM` trap is worth knowing: `ROWNUM` is assigned *before* `ORDER BY`, so `WHERE ROWNUM ⩽ 5 ORDER BY salary DESC` sorts five arbitrary rows instead of finding the top five.

LIMIT with WHERE and DISTINCT

Because `LIMIT` is last, it applies to whatever the rest of the query produced:

```sql
-- The three cheapest products that are in stock
SELECT name, price
FROM products
WHERE stock > 0
ORDER BY price, id
LIMIT 3;

-- The three highest distinct prices
SELECT DISTINCT price
FROM products
ORDER BY price DESC
LIMIT 3;
```

In the second query, duplicates are removed **before** the limit: if three products cost 900, the result is still three different prices. That ordering — distinct, then sort, then limit — is the heart of the classic "Nth highest salary" question in the top-N lesson.

LIMIT is not a performance switch by itself

`LIMIT 10` can make a query cheap: if an index already delivers rows in the `ORDER BY` order, the database reads ten rows and stops. Without such an index it must find and sort every matching row before it can know which ten come first, so `ORDER BY` + `LIMIT 10` on a million unindexed rows still sorts a million rows. And `OFFSET 100000` reads and throws away a hundred thousand rows before returning anything. The next lesson, on pagination, is about what to do instead.

Before you move on

- `LIMIT n` keeps at most `n` rows; `OFFSET m` skips `m` first; both run last.
- Always pair them with an `ORDER BY` whose last key is unique.
- MySQL's `LIMIT m, n` is offset first, count second.
- SQL Server uses `TOP` and `OFFSET … FETCH`; the standard and Oracle use `FETCH FIRST n ROWS ONLY`.
- `DISTINCT` happens before `LIMIT`.

EXERCISE 1

The three latest orders

The `orders` table:

id	customer	placed_on	total
1	Anita	2024-03-01	500
2	Ravi	2024-03-04	1200
3	Sam	2024-03-02	90
4	Lena	2024-03-04	300
5	Omar	2024-02-27	75

Return the three most recent orders as `id`, `customer`, `placed_on`, newest first. Orders placed on the same day: the higher `id` first. If there are fewer than three orders, return them all.

```
id | customer | placed_on
4 | Lena | 2024-03-04
2 | Ravi | 2024-03-04
3 | Sam | 2024-03-02
```

INPUT 1

```
CREATE TABLE orders (id INT PRIMARY KEY,
customer VARCHAR(40), placed_on DATE,
total INT);
INSERT INTO orders VALUES (1, 'Anita',
'2024-03-01', 500), (2, 'Ravi', '2024-
03-04', 1200), (3, 'Sam', '2024-03-02',
90), (4, 'Lena', '2024-03-04', 300), (5,
'Omar', '2024-02-27', 75);
```

OUTPUT 1

```
id | customer | placed_on
4 | Lena | 2024-03-04
2 | Ravi | 2024-03-04
3 | Sam | 2024-03-02
```

REFERENCE SOLUTION

```sql
SELECT id, customer, placed_on
FROM orders
ORDER BY placed_on DESC, id DESC
LIMIT 3;
```

EXERCISE 2

Rows four to six

The `products` table:

id	name	price
1	pen	10
2	ruler	25
3	glue	30
4	tape	30
5	folder	45
6	marker	15
7	scissors	80
8	stapler	120

Sorted by `price` ascending, then `id`, return only the **4th, 5th and 6th** rows as `id`, `name`, `price`. With fewer rows, return whatever exists in those positions.

```
id | name | price
3 | glue | 30
4 | tape | 30
5 | folder | 45
```

INPUT 1

```
CREATE TABLE products (id INT PRIMARY
KEY, name VARCHAR(40), price INT);
INSERT INTO products VALUES (1, 'pen',
10), (2, 'ruler', 25), (3, 'glue', 30),
(4, 'tape', 30), (5, 'folder', 45), (6,
'marker', 15), (7, 'scissors', 80), (8,
'stapler', 120);
```

OUTPUT 1

```
id | name | price
3 | glue | 30
4 | tape | 30
5 | folder | 45
```

REFERENCE SOLUTION

```
SELECT id, name, price
FROM products
ORDER BY price, id
LIMIT 3 OFFSET 3;
```

QUIZ

1. In MySQL, which rows does `LIMIT 10, 5` return?

A. Rows 10 to 15

B. Rows 11 to 15

C. Rows 6 to 15

D. Rows 11 to 20

2. What does `SELECT id FROM orders LIMIT 5` return?

A. The five oldest orders

B. The five newest orders

C. The five orders with the lowest id

D. Some five orders, in no guaranteed order

3. A query produces 12 rows. What does adding `LIMIT 5 OFFSET 20` return?

A. An error

B. The last 5 rows

C. No rows

D. All 12 rows

4. Which is the SQL Server way to return the first 5 rows by price?

A. `SELECT TOP (5) name FROM products ORDER BY price`

B. `SELECT name FROM products ORDER BY price LIMIT 5`

C. `SELECT name FROM products WHERE ROWNUM ⩽ 5`

D. `SELECT FIRST 5 name FROM products`

5. Three products cost 900, one 700 and one 500. What does `SELECT DISTINCT price FROM products ORDER BY price DESC LIMIT 2` return?

A. 900, 900

B. 900, 700

C. 900, 900, 900

D. 700, 500

ANSWER KEY · LIMIT AND OFFSET — TAKING A SLICE OF THE RESULT

1. B The comma form is offset first, count second: skip 10, take 5 — rows 11 to 15. It equals `LIMIT 5 OFFSET 10`. Reading it as "10 rows then 5" is the trap.

2. D LIMIT takes the first rows of the result's order, and without ORDER BY there is none. It may happen to return the lowest ids today and different rows tomorrow.

3. C An offset past the end is not an error; the slice is simply empty. That is how a paging client learns it went past the last page.

4. A SQL Server uses `TOP` (and `OFFSET … FETCH NEXT … ROWS ONLY` for slices); it does not support LIMIT. `ROWNUM` is old Oracle, and it is assigned before ORDER BY runs.

5. B DISTINCT removes duplicates before ORDER BY and LIMIT run, so the candidates are 900, 700, 500 and the first two are 900 and 700.

LESSON 4 · 15 MIN

Pagination — offset pages and keyset pages

Every list that can grow — search results, an order history, a feed, an admin table — is shown a page at a time. Fetching ten thousand rows to display twenty wastes the database, the network and the browser, so the query itself should return just the page. SQL has no `PAGE` keyword; you build pages out of `ORDER BY` and `LIMIT`. There are two ways to do it, offset pagination and keyset pagination, and choosing between them is a favourite backend-interview question because it touches correctness, performance and API design at once. This lesson covers both, the formula for offsets, the duplicate-and-missing-rows bug, and when each approach is right.

Offset pagination

The direct approach numbers the pages and skips the earlier ones:

```sql
-- Page 3, ten rows a page
SELECT id, title, created_at
FROM posts
ORDER BY created_at DESC, id DESC
LIMIT 10 OFFSET 20;
```

For 1-based page numbers the offset is

```
offset = (page - 1) * page_size
```

so page 1 uses `OFFSET 0`, page 2 `OFFSET 10`, page 3 `OFFSET 20`. In application code the numbers are passed as parameters — `LIMIT ? OFFSET ?` — and validated first: a page below 1, a page size of 10,000 or a negative number should be rejected or clamped, not handed to the database.

The total number of pages is the row count divided by the page size, rounded up: 45 rows at 10 a page is 5 pages, the last holding 5 rows. Counting rows is `SELECT COUNT(*) …`, which arrives with aggregation in a later module; many applications skip the exact count and simply show "next" while the page came back full.

Offset pagination is easy to build and lets the user jump straight to page 40. It has two problems.

Problem one: an order that is not unique

Suppose posts are sorted only by `created_at` and three posts share the timestamp at the boundary between pages 1 and 2. The database may order those three differently for the query of page 1 and the query of page 2 — they are separate queries, and ties have no defined order. One post can then appear on both pages while another appears on neither. The user never sees the second post; a test that walks every page counts one row twice.

The fix is the rule from the multi-key lesson: **end the sort with a unique key**.

```
ORDER BY created_at DESC, id DESC
```

Now every row has exactly one position, and the pages partition the result cleanly — as long as the data does not change.

Problem two: data that changes, and deep pages

Between the request for page 1 and page 2, a new post arrives at the top. Every row shifts down by one, so the last row of page 1 becomes the first row of page 2 and is shown twice. A deleted row does the opposite: one row moves up into page 1 after you have already left it and is never seen. For a feed that updates constantly, offset pages are visibly wrong.

And they get slower the deeper you go. To answer `LIMIT 10 OFFSET 100000`, the database cannot jump to position 100,000 — positions only exist after sorting. It must produce the first 100,010 rows in order and throw 100,000 of them away. Page 1 is fast, page 10,000 is not, and a crawler walking every page makes the database do quadratic work in total.

Keyset pagination

Keyset pagination, also called **seek** or **cursor** pagination, stops counting rows. Instead it remembers where the last page ended and asks for the rows after that point:

```sql
-- First page
SELECT id, title, created_at
FROM posts
ORDER BY created_at DESC, id DESC
LIMIT 10;

-- Next page: the last row shown had created_at = '2024-05-02', id = 731
SELECT id, title, created_at
FROM posts
WHERE created_at < '2024-05-02'
   OR (created_at = '2024-05-02' AND id < 731)
ORDER BY created_at DESC, id DESC
LIMIT 10;
```

The `WHERE` says "strictly after the last row in this sort order": an earlier timestamp, or the same timestamp with a smaller id. Its shape mirrors the `ORDER BY` key for key — the first key strictly past, or the first key equal and the second strictly past. Get the comparison direction from the sort direction: with `DESC` keys "after" means `<`, with `ASC` keys it means `>`.

When all keys run in the same direction, MySQL, PostgreSQL and SQLite accept a shorter **row value** comparison that means the same thing:

```sql
WHERE (created_at, id) < ('2024-05-02', 731)
```

Row values compare left to right like the sort itself. They cannot express mixed directions (`score DESC, id ASC`), so for those write the expanded `OR` form.

With an index on `(created_at, id)`, the database seeks straight to the boundary and reads ten rows, so page 10,000 costs what page 1 costs. Inserted and deleted rows no longer shift anything: the next page always starts right after the last row the user actually saw.

What keyset pagination costs

You cannot jump to page 40, because a page is defined by the row before it — the interface becomes "next" and "previous" (or infinite scroll), and the API hands the client an opaque **cursor** holding the last row's sort values. The sort key must be unique (hence the id), and the columns must not be NULL, or the `<` comparisons silently drop rows. "Previous page" is the same query with the comparison and the sort flipped, then the page reversed for display.

Choosing

Situation	Use
Admin table, small data, "go to page 7"	Offset
Search results people rarely page deep into	Offset, with a maximum page
Feeds, timelines, infinite scroll	Keyset
APIs that export everything page by page	Keyset
Data that changes while people browse	Keyset

Both need the same foundation: an `ORDER BY` that ends in a unique key. Without it neither approach is correct.

Before you move on

- Offset pages: `LIMIT size OFFSET (page - 1) * size`.
- A non-unique sort makes rows repeat or vanish between pages; add the primary key.
- Deep offsets read and discard every skipped row.
- Keyset pages filter past the last row seen: `WHERE key < last OR (key = last AND id < last_id)`.
- Row values `(a, b) < (x, y)` work when every key has the same direction.

EXERCISE 1

Page three of the catalogue

The `books` table (two books share a title):

id	title
1	Dune
2	Emma
3	Ulysses
4	Beloved
5	Hamlet
6	Ivanhoe
7	Middlemarch
8	Dracula
9	Persuasion
10	Rebecca
11	Emma

The catalogue lists books by `title`, then `id`, **four per page**. Return **page 3** as `id`, `title`. A page past the end is empty.

```
id | title
9 | Persuasion
10 | Rebecca
3 | Ulysses
```

INPUT 1

```
CREATE TABLE books (id INT PRIMARY KEY,
title VARCHAR(60));
INSERT INTO books VALUES (1, 'Dune'),
(2, 'Emma'), (3, 'Ulysses'), (4,
'Beloved'), (5, 'Hamlet'), (6,
'Ivanhoe'), (7, 'Middlemarch'), (8,
'Dracula'), (9, 'Persuasion'), (10,
'Rebecca'), (11, 'Emma');
```

OUTPUT 1

```
id | title
9 | Persuasion
10 | Rebecca
3 | Ulysses
```

REFERENCE SOLUTION

```sql
SELECT id, title
FROM books
ORDER BY title, id
LIMIT 4 OFFSET 8;
```

EXERCISE 2

The page after the cursor

A leaderboard is shown with **keyset pagination**: ordered by `points` descending, then `id` ascending, three rows a page. The last row the player saw had `points = 80` and `id = 4`. The `scores` table:

id	player	points
1	ana	95
2	bo	80
3	cy	88
4	dee	80
5	ed	80
6	flo	70
7	gus	60
8	hal	80

Return the next page — the first three rows that come **after** (80, 4) in that order — as `id`, `player`, `points`. Filter on the cursor's values with `WHERE`; the hidden datasets hold different rows before the cursor, so a fixed `OFFSET` will not work.

```
id | player | points
5 | ed | 80
8 | hal | 80
6 | flo | 70
```

INPUT 1

```
CREATE TABLE scores (id INT PRIMARY KEY,
player VARCHAR(20), points INT);
INSERT INTO scores VALUES (1, 'ana',
95), (2, 'bo', 80), (3, 'cy', 88), (4,
'dee', 80), (5, 'ed', 80), (6, 'flo',
70), (7, 'gus', 60), (8, 'hal', 80);
```

OUTPUT 1

```
id | player | points
5 | ed | 80
8 | hal | 80
6 | flo | 70
```

REFERENCE SOLUTION

```sql
SELECT id, player, points
FROM scores
WHERE points < 80
   OR (points = 80 AND id > 4)
ORDER BY points DESC, id
LIMIT 3;
```

QUIZ

1. What is the OFFSET for page 5 with 20 rows a page (pages numbered from 1)?

A. 100

B. 80

C. 5

D. 120

2. A list is paginated with `ORDER BY created_at LIMIT 10 OFFSET …`, and several rows share a `created_at`. What can go wrong?

A. Nothing; ties keep insertion order

B. Tied rows can swap between page queries, so one appears twice and another never

C. MySQL refuses the query

D. The pages come back in reverse

3. Why does `LIMIT 20 OFFSET 500000` get slow?

A. LIMIT is always slow on large tables

B. The database must produce and discard the 500,000 rows before the page

C. OFFSET disables every index

D. It sorts the result twice

4. Sorted by `created_at DESC, id DESC`, the last row shown has `created_at = '2024-05-02'` and `id = 731`. Which WHERE gives the next keyset page?

A. `WHERE created_at < '2024-05-02' AND id < 731`

B. `WHERE created_at < '2024-05-02' OR (created_at = '2024-05-02' AND id < 731)`

C. `WHERE id > 731`

D. `WHERE created_at ⩽ '2024-05-02'`

5. What does keyset pagination give up compared with offset pagination?

A. Correctness when rows are inserted

B. Speed on deep pages

C. Jumping straight to an arbitrary page number

D. The ability to sort descending

ANSWER KEY · PAGINATION — OFFSET PAGES AND KEYSET PAGES

1. B offset = (page − 1) × size = 4 × 20 = 80. Page 1 has offset 0. 100 would be the offset of page 6.

2. B Each page is a separate query, and ties have no defined order, so the boundary can fall differently each time. Adding a unique key (`, id`) fixes it; SQL never promises insertion order for ties.

3. B Positions only exist after ordering, so the engine cannot jump to row 500,000; it walks there. Keyset pagination seeks to the boundary with an index instead. Indexes are still usable with OFFSET — they just do not remove the walk.

4. B "After" means an earlier timestamp, or the same timestamp with a smaller id. The AND version drops older rows that happen to have larger ids; `⩽` repeats the rows of the same day already shown; `id > 731` ignores the sort entirely.

5. C A keyset page is defined by the row before it, so "go to page 40" needs pages 1–39 first. It is the method that stays correct under inserts and fast on deep pages, and it sorts in either direction.

LESSON 5 · 15 MIN

Top-N queries and the Nth highest value

"Show me the five best-selling products." "Who are the three newest customers?" "What is the second highest salary?" Questions of this shape — the first or last few rows by some measure, or the value in one particular position — are called **top-N queries**, and they are among the most asked SQL questions in placement rounds. With `ORDER BY` and `LIMIT` you already have the tools; what this lesson adds is the details that separate a right answer from an almost-right one: duplicates, ties, empty results and what to return when the answer does not exist.

Top N and bottom N

Sort by the measure, best first, and keep N:

```sql
-- The five most expensive products
SELECT name, price
FROM products
ORDER BY price DESC, id
LIMIT 5;

-- The three newest customers
SELECT name, joined_on
FROM customers
ORDER BY joined_on DESC, id DESC
LIMIT 3;
```

"Bottom N" is the same with the direction flipped. Combine with `WHERE` to rank only the candidates you care about — the cheapest in-stock items, the latest paid orders — because the filter runs before the sort and the limit:

```sql
SELECT name, price
FROM products
WHERE stock > 0 AND category = 'stationery'
ORDER BY price, id
LIMIT 3;
```

Remember the NULL rule when ranking a nullable column: in MySQL an ascending sort puts NULLs first, so "the three smallest discounts" could return three NULLs. Exclude them with

`WHERE discount IS NOT NULL` or push them to the end with `ORDER BY discount IS NULL, discount`.

The single top row

`LIMIT 1` gives the row holding the maximum, with all its columns:

```
SELECT name, salary
FROM employees
ORDER BY salary DESC, id
LIMIT 1;
```

The aggregate `MAX(salary)`, which arrives in the aggregation module, gives the top *value*; `ORDER BY … LIMIT 1` gives the top *row*, which is usually what "who earns the most" asks. If two people share the top salary, `LIMIT 1` returns one of them — the tie-breaker `id` decides which, and you should say so in an interview.

The Nth highest value

The famous one: *write a query that returns the second highest salary*. Take the salaries, highest first, skip one, keep one:

```
SELECT DISTINCT salary
FROM employees
ORDER BY salary DESC
LIMIT 1 OFFSET 1;
```

Why `DISTINCT`? Consider salaries 9000, 9000, 7000, 5000. Without it, the sorted list is 9000, 9000, 7000, 5000 and the second row is 9000 — the same value as the highest. The question asks for the second highest *salary*, a value, so duplicates must collapse first: 9000, 7000, 5000, and the second is 7000. `DISTINCT` runs before `ORDER BY` and `LIMIT`, which is exactly what is needed.

For the third highest, skip two: `LIMIT 1 OFFSET 2`. In general the Nth highest is `OFFSET N - 1`, but in MySQL that offset must be written as a literal number (or a prepared-statement placeholder); `OFFSET N - 1` with an expression is a syntax error. Inside a stored function, the classic LeetCode answer computes `SET N = N - 1;` first and then uses `LIMIT 1 OFFSET N`.

Returning NULL when there is no answer

If every employee earns 9000, there is no second highest salary, and the query above returns **no rows**. Many problem statements ask for `NULL` in that case — one row whose value is NULL. The trick is to use the query as a **scalar subquery**: a query in parentheses used where a single value is expected.

```sql
SELECT (
  SELECT DISTINCT salary
  FROM employees
  ORDER BY salary DESC
  LIMIT 1 OFFSET 1
) AS second_highest;
```

second_highest
NULL

The outer `SELECT` has no `FROM` and always produces exactly one row. Its one column is the value of the inner query — and a scalar subquery that finds no row evaluates to NULL. When a second salary does exist, the same query returns it. Subqueries get a full module later; this one pattern is worth learning now because it appears in so many interviews.

Ties at the cut-off

`LIMIT 3` returns exactly three rows even when the fourth row has the same score as the third. Whether that is right depends on the question. "Show three products for the banner" — fine, break the tie by id. "Award a prize to the top three scores" — not fine, because the person on the fourth row tied for third and deserves one too.

MySQL has no `FETCH FIRST 3 ROWS WITH TIES`. The ways to include ties are:

- find the third-highest distinct value and keep every row at least that high — a subquery, `WHERE score ⩾ (SELECT DISTINCT score … LIMIT 1 OFFSET 2)`;
- rank the rows with `DENSE_RANK()` and keep ranks 1 to 3 — a window function.

Both arrive later in the course. In an interview, the valuable move is noticing the tie question at all and asking which behaviour is wanted.

Top N per group is a different problem

"The two highest-paid employees in *each* department" looks similar but cannot be done with `ORDER BY … LIMIT`, because `LIMIT` applies once to the whole result, not once per department. Sorting by `dept, salary DESC` puts each department's best at the top of its block, but cutting the blocks needs a window function (`ROW_NUMBER() OVER (PARTITION BY dept ORDER BY salary DESC)`) or a correlated subquery. Recognising that boundary is part of knowing top-N queries.

Before you move on

- Top N: `ORDER BY measure DESC, id LIMIT n`; bottom N flips the direction.
- Filter first, and decide what NULLs should do.
- Nth highest value: `SELECT DISTINCT … ORDER BY … DESC LIMIT 1 OFFSET n-1`, with a literal offset.
- Wrap it in `SELECT (…) AS name` to get NULL instead of no rows.
- `LIMIT` cuts ties arbitrarily and works on the whole result, never per group.

EXERCISE 1

The cheapest items in stock

The `products` table:

id	name	price	stock
1	pen	10	0
2	pencil	5	12
3	eraser	8	3
4	ruler	12	7
5	glue	8	0
6	tape	15	4

Return the **three cheapest products that are in stock** (`stock > 0`) as `name`, `price`, cheapest first, equal prices by `id`.

```
name | price
pencil | 5
eraser | 8
ruler | 12
```

INPUT 1

```
CREATE TABLE products (id INT PRIMARY KEY, name VARCHAR(40), price INT, stock INT);
INSERT INTO products VALUES (1, 'pen', 10, 0), (2, 'pencil', 5, 12), (3, 'eraser', 8, 3), (4, 'ruler', 12, 7), (5, 'glue', 8, 0), (6, 'tape', 15, 4);
```

OUTPUT 1

```
name | price
pencil | 5
eraser | 8
ruler | 12
```

REFERENCE SOLUTION

```sql
SELECT name, price
FROM products
WHERE stock > 0
ORDER BY price, id
LIMIT 3;
```

EXERCISE 2

Second highest salary

The `employees` table:

id	name	salary
1	Asha	9000
2	Ben	9000
3	Chen	7000
4	Dina	5000

Return the **second highest distinct salary** in one column named `second_highest`. If there is no second highest salary, return a single row holding `NULL`.

```
second_highest
7000
```

INPUT 1

```
CREATE TABLE employees (id INT PRIMARY
KEY, name VARCHAR(40), salary INT);
INSERT INTO employees VALUES (1, 'Asha',
9000), (2, 'Ben', 9000), (3, 'Chen',
7000), (4, 'Dina', 5000);
```

OUTPUT 1

```
second_highest
7000
```

REFERENCE SOLUTION

```sql
SELECT (
  SELECT DISTINCT salary
  FROM employees
  ORDER BY salary DESC
  LIMIT 1 OFFSET 1
) AS second_highest;
```

QUIZ

1. Salaries are 9000, 9000, 7000. What does `SELECT salary FROM employees ORDER BY salary DESC LIMIT 1 OFFSET 1` return?

A. 7000

B. 9000

C. NULL

D. No rows

2. Every employee earns 5000. What does `SELECT (SELECT DISTINCT salary FROM employees ORDER BY salary DESC LIMIT 1 OFFSET 1) AS s` return?

A. No rows

B. One row: NULL

C. One row: 5000

D. An error

3. Why is `SELECT DISTINCT salary FROM employees ORDER BY salary DESC LIMIT 1 OFFSET N - 1` not valid MySQL?

A. DISTINCT cannot be used with LIMIT

B. MySQL's LIMIT and OFFSET take literal numbers or placeholders, not expressions

C. OFFSET must come before LIMIT

D. N is a reserved word

4. Scores are 90, 85, 80, 80. What does `ORDER BY score DESC, id LIMIT 3` do with the two 80s?

A. Returns both, making four rows

B. Returns exactly one of them, chosen by id

C. Returns neither

D. Raises an error about ties

5. Which needs more than ORDER BY and LIMIT?

A. The five newest orders

B. The cheapest product in stock

C. The two best-paid employees in each department

D. The third highest distinct price

ANSWER KEY · TOP-N QUERIES AND THE NTH HIGHEST VALUE

1. B Without DISTINCT the sorted list is 9000, 9000, 7000, and the second row is the second 9000. Add DISTINCT to rank values rather than rows.

2. B The outer SELECT without FROM always yields one row, and a scalar subquery that finds no row evaluates to NULL. The inner query alone would return no rows.

3. B MySQL allows only non-negative integer literals or `?` placeholders there, which is why the LeetCode function answer does `SET N = N - 1` first. DISTINCT with LIMIT is fine.

4. B LIMIT always cuts at exactly three rows; the tie-breaker decides which 80 survives. Including ties needs `WITH TIES` (not in MySQL), a subquery or `DENSE_RANK()`.

5. C LIMIT applies once to the whole result, so it cannot keep two rows per department; that is top-N per group, solved with window functions. The others are single top-N or Nth-value queries.

CHECKPOINT · 25 MIN

Checkpoint — Sorting and limiting

This checkpoint covers the whole module: `ORDER BY` with `ASC` and `DESC`, sorting by aliases, expressions and positions, where NULLs sort in MySQL and in other databases, multi-key sorts and tie-breakers, custom orders with `CASE`, `LIMIT` and `OFFSET` (and MySQL's comma form), `TOP` and `FETCH FIRST` in other dialects, offset and keyset pagination, and top-N and Nth-highest queries.

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:

- Why does a query without `ORDER BY` have no guaranteed row order, and why does `LIMIT` need one?
- In `ORDER BY a, b DESC`, which keys are descending?
- Where does MySQL put NULLs in an ascending sort, and how do you move them to the end?
- Why can an alias appear in `ORDER BY` but not in `WHERE`?
- What do `LIMIT 5 OFFSET 10` and `LIMIT 5, 10` return?
- What is the offset of page `p` with `s` rows a page, and why do deep offsets get slow?
- How does keyset pagination find the next page?
- Why does the second-highest-salary query need `DISTINCT`, and how do you make it return NULL?

The three queries are a leaderboard sorted on several keys with a tie-breaker, a page from the middle of a catalogue with NULL prices sent to the end, and the third highest distinct score — NULL when there is none.

EXERCISE 1

Leaderboard

The `players` table:

id	name	points	joined_on
1	Asha	120	2024-01-05
2	Ben	150	2024-02-01
3	Chen	120	2024-01-02
4	Dina	90	2024-01-01
5	Eli	150	2024-02-01
6	Fay	60	2023-12-30
7	Gus	120	2024-01-05

Return the **top five** players as `name`, `points`: most points first; on equal points, the player who joined earlier first; on the same day, the lower `id` first.

```
name | points
Ben | 150
Eli | 150
Chen | 120
Asha | 120
Gus | 120
```

INPUT 1

```
CREATE TABLE players (id INT PRIMARY
KEY, name VARCHAR(40), points INT,
joined_on DATE);
INSERT INTO players VALUES (1, 'Asha',
120, '2024-01-05'), (2, 'Ben', 150,
'2024-02-01'), (3, 'Chen', 120, '2024-
01-02'), (4, 'Dina', 90, '2024-01-01'),
(5, 'Eli', 150, '2024-02-01'), (6,
'Fay', 60, '2023-12-30'), (7, 'Gus',
120, '2024-01-05');
```

OUTPUT 1

```
name | points
Ben | 150
Eli | 150
Chen | 120
Asha | 120
Gus | 120
```

REFERENCE SOLUTION

```sql
SELECT name, points
FROM players
ORDER BY points DESC, joined_on, id
LIMIT 5;
```

EXERCISE 2

Catalogue page, unpriced items last

The `items` table; `price` is NULL for items not priced yet:

id	name	price
1	lamp	NULL
2	desk	4500
3	chair	1200
4	mat	300
5	shelf	NULL
6	bin	300
7	rug	2500

The catalogue sorts by `price` ascending with **NULL prices at the end**, then by `id`, **three items a page**. Return **page 2** as `id`, `name`, `price`.

```
id | name | price
7 | rug | 2500
2 | desk | 4500
1 | lamp | NULL
```

INPUT 1

```
CREATE TABLE items (id INT PRIMARY KEY,
name VARCHAR(40), price INT);
INSERT INTO items VALUES (1, 'lamp',
NULL), (2, 'desk', 4500), (3, 'chair',
1200), (4, 'mat', 300), (5, 'shelf',
NULL), (6, 'bin', 300), (7, 'rug',
2500);
```

OUTPUT 1

```
id | name | price
7 | rug | 2500
2 | desk | 4500
1 | lamp | NULL
```

REFERENCE SOLUTION

```
SELECT id, name, price
FROM items
ORDER BY price IS NULL, price, id
LIMIT 3 OFFSET 3;
```

EXERCISE 3

Third highest distinct score

The `scores` table:

id	player	score
1	ana	88
2	bo	95
3	cy	88
4	dee	70
5	ed	95
6	flo	60

Return the **third highest distinct score** in one column named `third_highest` — a single row holding `NULL` when there are fewer than three distinct scores.

```
third_highest
70
```

INPUT 1

```
CREATE TABLE scores (id INT PRIMARY KEY,
player VARCHAR(20), score INT);
INSERT INTO scores VALUES (1, 'ana',
88), (2, 'bo', 95), (3, 'cy', 88), (4,
'dee', 70), (5, 'ed', 95), (6, 'flo',
60);
```

OUTPUT 1

```
third_highest
70
```

REFERENCE SOLUTION

```
SELECT (
  SELECT DISTINCT score
  FROM scores
  ORDER BY score DESC
  LIMIT 1 OFFSET 2
) AS third_highest;
```

CHECKPOINT QUESTIONS

1. Which clause runs last in the logical order of a query?

A. WHERE

B. SELECT

C. ORDER BY

D. LIMIT

2. `ORDER BY dept DESC, name` sorts…

A. Both keys descending

B. dept descending, name ascending

C. dept ascending, name descending

D. Only by dept

3. In PostgreSQL, `ORDER BY bonus` (ascending) puts NULL bonuses…

A. First

B. Last

C. Nowhere; they are removed

D. Randomly

4. What does this return in MySQL?

```sql
SELECT name FROM t ORDER BY score IS NULL, score DESC;
```

A. NULL scores first, then highest score first

B. Highest score first, NULL scores last

C. Lowest score first, NULL scores last

D. A syntax error

5. `SELECT DISTINCT name FROM employees ORDER BY hire_date` is refused by MySQL. Why?

A. DISTINCT cannot be combined with ORDER BY

B. With DISTINCT, the sort key must be in the select list, since one name may have several hire dates

C. hire_date must be indexed

D. ORDER BY needs ASC or DESC after DISTINCT

6. What does MySQL's `LIMIT 20, 10` return?

A. Rows 21–30

B. Rows 11–30

C. Rows 21–40

D. Rows 20–30

7. Which is the standard SQL way (also PostgreSQL and Oracle 12c+) to take the first 10 rows?

A. `TOP 10`

B. `FETCH FIRST 10 ROWS ONLY`

C. `WHERE ROWNUM ⩽ 10`

D. `LIMIT 10` is the standard

8. What is the offset of page 4 at 25 rows a page?

A. 100

B. 75

C. 4

D. 96

9. A feed uses offset pagination and new posts arrive constantly. What does a reader moving from page 1 to page 2 see?

A. Exactly the next ten posts

B. Some posts from page 1 again, because new posts pushed every row down

C. An error

D. Only the new posts

10. Why must the ORDER BY used for keyset pagination end with a unique column?

A. So the WHERE condition identifies one exact boundary and no tied rows are skipped or repeated

B. Because MySQL requires a primary key in ORDER BY

C. To make the query use LIMIT

D. It does not need to

11. Salaries are 8000, 6000, 6000, 4000. What does `SELECT DISTINCT salary FROM e ORDER BY salary DESC LIMIT 1 OFFSET 2` return?

A. 6000

B. 4000

C. NULL

D. No rows

12. Which query returns a single NULL row when there is no second highest salary?

A. `SELECT DISTINCT salary FROM e ORDER BY salary DESC LIMIT 1 OFFSET 1`

B. `SELECT (SELECT DISTINCT salary FROM e ORDER BY salary DESC LIMIT 1 OFFSET 1) AS s`

C. `SELECT salary FROM e ORDER BY salary DESC LIMIT 2`

D. `SELECT DISTINCT salary FROM e WHERE salary IS NULL`

ANSWER KEY · CHECKPOINT — SORTING AND LIMITING

1. D FROM → WHERE → SELECT → ORDER BY → LIMIT. The slice is taken after the rows are filtered, computed and sorted, which is why LIMIT needs ORDER BY to be meaningful.

2. B Each key has its own direction, and ascending is the default. DESC applies only to `dept`.

3. B PostgreSQL and Oracle treat NULL as larger than every value, so NULLs are last ascending — the opposite of MySQL. PostgreSQL also accepts `NULLS FIRST` / `NULLS LAST`; MySQL does not.

4. B `score IS NULL` is 0 for values and 1 for NULLs, so values come first; within them `score DESC` puts the highest first. (A plain `score DESC` would also put NULLs last in MySQL — the prefix makes it explicit and portable.)

5. B After duplicates are removed, a row may stand for several hire dates and the database cannot choose one to sort by. Without DISTINCT, sorting by an unselected column is fine.

6. A Offset first, then count: skip 20, take 10 — rows 21 to 30, the same as `LIMIT 10 OFFSET 20`.

7. B `OFFSET … ROWS FETCH FIRST|NEXT … ROWS ONLY` is the SQL standard. `LIMIT` is a MySQL/PostgreSQL/SQLite extension, `TOP` is SQL Server, `ROWNUM` is old Oracle.

8. B (4 − 1) × 25 = 75. Page 1 starts at offset 0.

9. B Each insert at the top shifts every offset by one, so the end of page 1 becomes the start of page 2. Keyset pagination starts after the last row actually seen and does not shift.

10. A The cursor is the last row's sort values. If several rows share them, `> cursor` skips the unseen tied rows and `⩾ cursor` repeats the seen ones. A unique last key makes the boundary exact.

11. B The distinct values are 8000, 6000, 4000; skipping two leaves 4000. Without DISTINCT the third row would be 6000.

12. B The outer SELECT always yields one row, and the empty scalar subquery becomes NULL. The bare query returns no rows; the others answer different questions.

CodeKairo · SQL study plan · Module 3: Sorting and limiting. Solve the exercises online to track your progress and earn XP.