Module 1 of 16

Foundations

what a relational database is — tables, typed columns, rows without order, primary and foreign keys; SQL as a declarative language and its statement families; MySQL beside PostgreSQL, SQLite and SQL Server; the first SELECT; and the logical order a query's clauses run in.

Before the first query it pays to know what you are talking to. A relational database keeps data in tables — named, typed columns and a set of rows with no order of their own — and ties tables together with keys: a primary key that identifies each row, foreign keys that point at another table's rows so each fact is stored once. The database management system underneath stores those tables, checks every write against their rules, answers queries through indexes, runs transactions and keeps many users out of each other's way. SQL is the language for all of it: declarative, so a statement describes the rows it wants and the optimizer decides how to find them.

What a database is covers tables, rows and columns, schema and data, primary and foreign keys, the DBMS's jobs and why a spreadsheet is not one. What SQL is covers declarative versus imperative, the statement families DDL, DML, DQL, DCL and TCL, statements and scripts, case, quotes and comments. MySQL and other databases covers the family — MySQL, MariaDB, PostgreSQL, SQLite, SQL Server, Oracle — and the seams between their dialects: `LIMIT` versus `TOP` versus `FETCH FIRST`, `CONCAT` versus `||`, quoting, auto-numbered ids, NULL helpers, dates and case-insensitive comparison. Your first SELECT covers choosing columns, `SELECT *` and its costs, `ORDER BY` for a dependable order, `SELECT` without a table and literals beside columns. How a query runs covers the logical order of the clauses — FROM, WHERE, GROUP BY, HAVING, SELECT, DISTINCT, ORDER BY, LIMIT — what it means for aliases, and the parser, optimizer and executor behind it.

The course teaches MySQL 8, and every exercise is a query judged against tables that a setup script builds — the input you see beside the editor. The exercises read whole tables in a fixed order, count rows, insert a row and read the result, run two statements as one script, build full names with `CONCAT`, take a top three with `LIMIT`, choose and reorder columns, and filter on an expression while sorting by its alias. The checkpoint asks for a table read back in order, a report with literal and computed columns, and a filtered list sorted on a computed value with a tie-breaker.

CONTENTS

1	What a database is — tables, rows and keys	13 min
2	What SQL is — a declarative language in five parts	13 min
3	MySQL and its cousins — where the dialects differ	14 min
4	Your first SELECT — columns, tables and order	13 min
5	How a query runs — the logical order of the clauses	14 min
✓	Checkpoint — Foundations	25 min

Exercises in this module compile on MySQL 8 syntax on SQLite 3.49. Reference solutions and the answer key follow each lesson.

LESSON 1 · 13 MIN

What a database is — tables, rows and keys

Almost every application you have used keeps its data in a database: the bank that holds your balance, the shop that remembers your orders, the platform that records which problems you have solved. Before writing a line of SQL it pays to know what you are talking to — what a table is, what a row and a column mean, what a key is for and what the software underneath actually does. This lesson builds that picture, and the vocabulary every later lesson and every interview assumes.

Data as tables

A **relational database** stores data in **tables**. A table has a name and a fixed list of **columns**; each column has a name and a **type**. The data itself is a set of **rows**, each row holding one value per column.

A table: a fixed heading over a set of rows

students

id	name	year	city
1	Asha	2	Pune
2	Ravi	1	Chennai
3	Meera	3	Pune

id: unique, never NULL — the primary key

1. A relational table has a name, students, a fixed list of columns and a set of rows. This one holds 3 rows; the database keeps them in no particular order.
2. The heading — each column's name and type — is the schema. It changes rarely and on purpose; rows are inserted, updated and deleted all day.
3. Read across and a row is one record: everything the table knows about one student, here Ravi in year 1 from Chennai.
4. Read down and a column is one attribute of every row. A query names it, city, never "the fourth column", and every value in it has the column's type.
5. id has a different value in every row and none is NULL, so it identifies exactly one row: the primary key. Two students may share a name and a city, never an id.

Read it the way the database does: one row is one **record** (one student), one column is one **attribute** (every student's city). The column list and the types are the table's **schema**; the rows

are its **data**. The schema changes rarely and deliberately; rows are inserted, updated and deleted all day.

Three properties matter more than they first appear:

- **Every value in a column has the column's type.** `year` is an integer, `city` is text. The database refuses `'second'` in `year` (or, in a lax mode, converts it) — a spreadsheet cell would accept anything.
- **Rows have no inherent order.** A table is a *set* of rows, not a list. The order you see when you read a table is whatever the engine found convenient; if order matters you ask for it with `ORDER BY`, every time.
- **Columns are named, not numbered.** A query says `city`, never "the fourth column".

The word *relational* comes from mathematics: a table is a *relation*, a set of tuples that share a heading. You do not need the theory, but it explains the habits — sets have no duplicates and no order, and SQL behaves accordingly unless you ask otherwise.

Keys identify rows

If two students are both called Ravi and both live in Chennai, how do you update one of them? You need a column whose value is different in every row. That is a **key**.

The **primary key** is the key a table chooses as its identity. It must be **unique** (no two rows share a value) and **never NULL** (every row has one). The usual choice is an integer `id` that the database assigns itself:

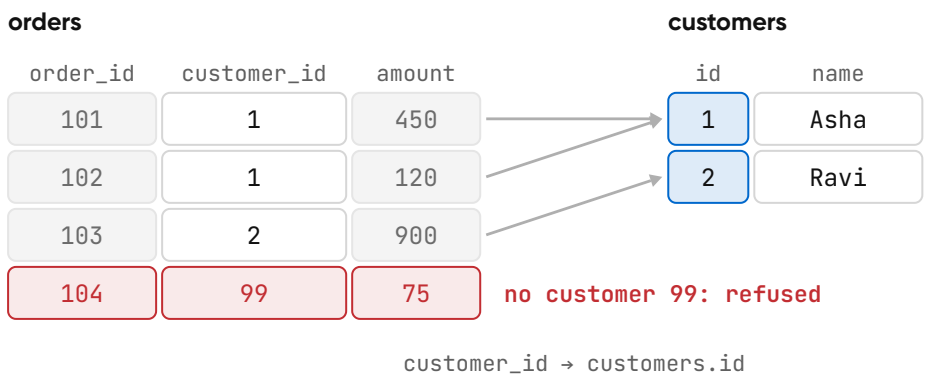
```sql
CREATE TABLE students (
  id   INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(50) NOT NULL,
  year INT NOT NULL,
  city VARCHAR(40)
);
```

Do not worry about the syntax yet — `CREATE TABLE` gets a module of its own. Notice the shape: a name and a type per column, plus rules (`PRIMARY KEY`, `NOT NULL`) that the database will check on every insert. A natural value such as an email address can also be a key; a separate number is preferred because names and emails change, and a key that changes has to change everywhere it is referenced.

Foreign keys relate tables

Real data is spread across several tables, each about one kind of thing. An order belongs to a customer, so the `orders` table stores the customer's key:

A foreign key points at another table's primary key



1. Each order stores only its customer's id. Every customer_id is a value of customers.id, so each order points at exactly one customer — 2 of them at Asha.
2. The name is stored once, in customers. Rename Asha and both of her orders read the new name through the pointer; nothing was copied, so nothing is left stale.
3. Order 104 names customer 99, and no row of customers has that id. With the foreign key declared, the database refuses the insert instead of storing an order for nobody.

`orders.customer_id` is a **foreign key**: its values must be values of `customers.id`. The customer's name is stored **once**, in `customers`; an order only points at it. Change Asha's name and every one of her orders is up to date, because none of them copied it. That is the core idea of relational design — store each fact once and connect facts by keys — and joins, later in the course, are how a query follows those pointers back.

Declared as a constraint, the foreign key lets the database refuse an order for customer 99 when no customer 99 exists, and refuse to delete a customer who still has orders.

What the DBMS does

The tables are the **database**. The software that keeps them is the **database management system** (DBMS) — MySQL, PostgreSQL, SQLite, SQL Server, Oracle. People say "the database" for both; interviewers sometimes check that you know the difference. The DBMS:

- **stores** the tables on disk in pages and caches the hot ones in memory;

- **enforces** the schema — types, `NOT NULL`, `UNIQUE`, primary and foreign keys, `CHECK` rules — on every write;
- **answers queries**: you describe the rows you want and it works out how to find them, using **indexes** (sorted lookup structures) so it does not read every row;
- **runs transactions**: a group of changes either all happen or none do, even if the power fails halfway;
- **handles concurrency**: hundreds of connections read and write at once without seeing each other's half-finished work;
- **controls access**: users, passwords and permissions on each table.

Most DBMSs are **servers**: a program running on a machine, which your application connects to over the network. SQLite is the exception — a library inside your program that keeps the whole database in one file.

Why not a spreadsheet or a file?

A spreadsheet looks like a table and a CSV file holds rows, so why the extra machinery? Because the machinery is the point:

Need	Spreadsheet / CSV	Database
A wrong type in a column	accepted silently	refused
Two people editing at once	last save wins	both changes kept, conflicts locked
Crash in the middle of a change	half-written file	rolled back
Find one row among 10 million	read everything	index lookup
The same customer in 500 orders	copied 500 times	stored once, referenced
Who may read salaries	whoever has the file	per-user permissions

A spreadsheet is fine for one person and a few thousand rows. The moment data is shared, large or valuable, those rows of the table become requirements.

Relational and the rest

Relational databases are not the only kind. **Document stores** (MongoDB) keep JSON-like documents; **key-value stores** (Redis) map keys to values in memory; **wide-column**, **graph** and **time-series** databases each optimise one access pattern. They are often grouped as "NoSQL". They trade the relational model's guarantees — a fixed schema, joins, multi-row transactions — for scale or flexibility in one direction. Relational databases remain the default for business data, and SQL is the language nearly all of them share; several NoSQL systems have added SQL-like query languages because people know it.

A first look at a table

In every exercise of this course the tables are built for you by a **setup script**, shown as the exercise's input, and your SQL runs against them. The simplest question you can ask a table is "show me everything":

```sql
SELECT * FROM students ORDER BY id;
```

id	name	year	city
1	Asha	2	Pune
2	Ravi	1	Chennai
3	Meera	3	Pune

`*` means every column, `FROM students` names the table, and `ORDER BY id` fixes the order of the rows — remember, a table has none of its own. The other question you can ask right away is how many rows there are:

```sql
SELECT COUNT(*) AS total FROM students;
```

total
3

`COUNT(*)` counts rows and `AS total` names the result column. Both are explained properly later; for now they let you look at data.

Key takeaways

- A relational database stores **tables**; a table has typed, named **columns** and a set of **rows** with **no inherent order**.
- A **primary key** identifies each row: unique and never NULL. A **foreign key** points at another table's primary key and records a relationship.
- Store each fact once and connect facts by keys.
- The **DBMS** (MySQL, PostgreSQL, SQLite…) stores data, enforces rules, answers queries with indexes, runs transactions and handles many users at once.
- `SELECT * FROM t ORDER BY id` shows a whole table; `ORDER BY` is how you get an order.

EXERCISE 1

Show the whole table

The table `books` holds a small library catalogue:

book_id	title	author	year
3	Clean Code	Robert Martin	2008
1	The Pragmatic Programmer	Andy Hunt	1999
2	Refactoring	Martin Fowler	1999

Return **every column** of every book, ordered by `book_id` ascending. The rows were inserted out of order, so the order must come from your query.

Output: the columns `book_id | title | author | year`, one row a book, by `book_id`.

```
book_id | title | author | year
1 | The Pragmatic Programmer | Andy Hunt | 1999
2 | Refactoring | Martin Fowler | 1999
3 | Clean Code | Robert Martin | 2008
```

INPUT 1

```
CREATE TABLE books (
  book_id INT PRIMARY KEY,
  title   VARCHAR(100) NOT NULL,
  author  VARCHAR(60),
  year    INT
);
INSERT INTO books VALUES
  (3, 'Clean Code', 'Robert Martin',
2008),
  (1, 'The Pragmatic Programmer', 'Andy
Hunt', 1999),
  (2, 'Refactoring', 'Martin Fowler',
1999);
```

OUTPUT 1

```
book_id | title | author | year
1 | The Pragmatic Programmer | Andy Hunt
| 1999
2 | Refactoring | Martin Fowler | 1999
3 | Clean Code | Robert Martin | 2008
```

REFERENCE SOLUTION

```sql
SELECT *
FROM books
ORDER BY book_id;
```

EXERCISE 2

Count the rows

Using the same `books` table, return how many books the catalogue holds, as one column named `total_books`.

book_id	title	author	year
3	Clean Code	Robert Martin	2008
1	The Pragmatic Programmer	Andy Hunt	1999
2	Refactoring	Martin Fowler	1999

Output: one row, one column `total_books`.

```
total_books
3
```

INPUT 1

```
CREATE TABLE books (
  book_id INT PRIMARY KEY,
  title   VARCHAR(100) NOT NULL,
  author  VARCHAR(60),
  year    INT
);
INSERT INTO books VALUES
  (3, 'Clean Code', 'Robert Martin',
2008),
  (1, 'The Pragmatic Programmer', 'Andy
Hunt', 1999),
  (2, 'Refactoring', 'Martin Fowler',
1999);
```

OUTPUT 1

```
total_books
3
```

REFERENCE SOLUTION

```sql
SELECT COUNT(*) AS total_books
FROM books;
```

QUIZ

1. In a relational table, what does one **row** represent?

A. One attribute shared by every record

B. One record — for example one student

C. The table's schema

D. One index entry

2. Which two properties must a **primary key** have?

A. Unique and indexed

B. Unique and never NULL

C. Numeric and auto-incremented

D. Not NULL and pointing at another table

3. `orders.customer_id` holds values of `customers.id`. What is `orders.customer_id`?

A. A primary key

B. A foreign key

C. An index

D. A duplicate of the customer's data

4. You run `SELECT * FROM students` twice and get the rows in the same order. What does SQL guarantee about the order next time?

A. The same order, always

B. Insertion order

C. Primary-key order

D. Nothing — only `ORDER BY` fixes an order

5. Which job belongs to the **DBMS** rather than to the data?

A. Holding the students' names

B. Refusing an insert that duplicates a primary key

C. Being a table

D. Having columns

ANSWER KEY · WHAT A DATABASE IS — TABLES, ROWS AND KEYS

1. B A row is one record with a value for each column. A column is one attribute across all records; the schema is the column list and types.

2. B A primary key identifies exactly one row, so its value must be unique and present in every row. It is usually an auto-numbered integer, but it need not be; pointing at another table is what a foreign key does.

3. B A foreign key refers to another table's primary key. It stores the relationship, not a copy of the customer's details — those stay in `customers`, stored once.

4. D A table is a set of rows with no inherent order. The engine returns rows in whatever order is cheapest, which can change after inserts, deletes or a new index. Only `ORDER BY` guarantees one.

5. B The database is the data; the DBMS is the software that stores it, enforces constraints, runs queries and transactions, and manages concurrent users.

LESSON 2 · 13 MIN

What SQL is — a declarative language in five parts

SQL — Structured Query Language, said "S-Q-L" or "sequel" — is how you talk to a relational database. It was designed at IBM in the 1970s, standardised by ANSI and ISO from 1986, and is still the one language that MySQL, PostgreSQL, SQLite, SQL Server and Oracle all speak. Each speaks its own **dialect**, but the core you learn in this course carries across all of them. This lesson is about what kind of language SQL is, the families its statements fall into, and the small syntax rules that trip people up in their first week.

Declarative: describe the result

Most programming languages are **imperative**: you tell the computer the steps. To find the students who live in Pune in Python you write the loop:

```
result = []
for s in students:
    if s.city == "Pune":
        result.append(s.name)
```

SQL is **declarative**: you describe the result and the database finds the steps.

```
SELECT name
FROM students
WHERE city = 'Pune';
```

Nothing in that query says how to find the rows. If there is an index on `city`, the database uses it and touches only the matching rows; if not, it reads the table. If the table moves to a faster disk, gains a million rows or is split across machines, the query does not change. The part of the database that picks the method is the **optimizer**, and the method it picks is the **execution plan**. That division of labour is why a five-line query can replace fifty lines of loops — and why, later in the course, writing queries the optimizer can serve well becomes a topic of its own.

The flip side: SQL thinks in **sets of rows**, not one row at a time. A `WHERE` condition is a test applied to every row; an `UPDATE` changes every row that matches. Reasoning "for each row…"

is fine for understanding, but the statement acts on the whole set at once.

The statement families

Every SQL statement belongs to one of a few families. Interviews ask for the names, and the names help you read documentation:

Family	Stands for	Statements	Acts on
DDL	Data Definition Language	`CREATE`, `ALTER`, `DROP`, `TRUNCATE`, `RENAME`	the structure: tables, indexes, views
DML	Data Manipulation Language	`INSERT`, `UPDATE`, `DELETE`	the rows
DQL	Data Query Language	`SELECT`	reading rows
DCL	Data Control Language	`GRANT`, `REVOKE`	permissions
TCL	Transaction Control Language	`COMMIT`, `ROLLBACK`, `SAVEPOINT`	transactions

Some books fold `SELECT` into DML; both are accepted answers. Here is one statement from each of the first three, in the order a new table meets them:

```sql
-- DDL: make the table
CREATE TABLE tasks (
  id    INT PRIMARY KEY,
  title VARCHAR(100) NOT NULL,
  done  INT NOT NULL DEFAULT 0
);

-- DML: put rows in, change them
INSERT INTO tasks (id, title) VALUES (1, 'Buy milk'), (2, 'Call Ravi');
UPDATE tasks SET done = 1 WHERE id = 1;

-- DQL: read them back
SELECT id, title, done FROM tasks ORDER BY id;
```

id	title	done
1	Buy milk	1
2	Call Ravi	0

Most of this course is DQL — `SELECT` is where the depth is. DML and DDL get their own modules later, and TCL appears with transactions.

Statements and scripts

A **statement** is one complete command. Statements end with a **semicolon**; the semicolon separates them when several are sent together as a **script**. In the exercises your answer is such a script — usually one `SELECT`, sometimes an `INSERT` followed by the `SELECT` that shows its effect. Each `SELECT` prints its result: the column names joined by `|`, then one line a row. A statement that returns no rows (an `INSERT`, an `UPDATE`) prints nothing.

Whitespace and line breaks do not matter. These are the same statement:

```
SELECT id, title FROM tasks WHERE done = 0;

SELECT id,
       title
FROM   tasks
WHERE  done = 0;
```

The second form — one clause per line, keywords aligned — is how real code is written. It makes the clauses easy to find and the diff easy to read.

Case

Keywords are not case sensitive. `SELECT`, `select` and `SeLeCt` are the same keyword. The convention is capitals for keywords and lower case for names, so the structure stands out; this course follows it.

Names (tables and columns) depend on the database. In MySQL, column names are never case sensitive, and table names follow the file system unless configured otherwise — case sensitive on Linux, not on Windows. PostgreSQL folds unquoted names to lower case. The safe

habit: name everything in lower case with underscores (`order_items`, `created_at`) and spell it the same way everywhere.

Values are another matter. Whether `'pune' = 'Pune'` is true depends on the column's **collation** — the rules for comparing text. MySQL's default collations are case-insensitive, so it is true there; PostgreSQL compares case-sensitively by default, so it is false. When case matters to the answer, don't rely on either: store data consistently.

Quotes

- **Strings** go in **single quotes**: `'Pune'`, `'2024-03-01'`. A single quote inside a string is written twice: `'it''s'`. MySQL also accepts a backslash escape, `'it\'s'`, which other databases do not.
- **Identifiers** that need quoting — a name with a space, or one that clashes with a keyword — use **backticks** in MySQL (`` `order` ``, `` `full name` ``) and **double quotes** in standard SQL and PostgreSQL (`"order"`).
- MySQL also accepts **double-quoted strings** by default. Don't use them: in PostgreSQL and SQLite `"Pune"` means a *column* called Pune, and the query fails or, worse, compares a column with itself.
- **Numbers** are never quoted: `WHERE year = 2`, not `WHERE year = '2'`. MySQL would convert the string, but the habit hides type mistakes.

Comments

```sql
-- a line comment: two dashes and a space, to the end of the line
# MySQL also takes a hash for a line comment
/* a block comment
   can span lines */
SELECT title FROM tasks;  -- after a statement too
```

Use `--` for portability; `#` is MySQL only. In this course the starter of every exercise is a comment describing the task — a script with nothing but comments has no statement, which is why running it says to write a query first.

SQL is a standard with dialects

The ISO standard defines the core — `SELECT`, joins, `GROUP BY`, subqueries, window functions, transactions — and each database implements most of it plus extensions of its own. String concatenation, limiting rows, auto-numbered ids, date arithmetic and the names of many functions differ. The next lesson walks through the differences interviewers ask about. The rule for now: learn one dialect properly (this course uses MySQL 8) and know where the seams are.

Key takeaways

- SQL is **declarative**: describe the result; the **optimizer** chooses the method.
- It works on **sets of rows**: a condition tests every row, a change hits every matching row.
- Families: **DDL** (structure), **DML** (rows), **DQL** (`SELECT`), **DCL** (permissions), **TCL** (transactions).
- Statements end with `;`; whitespace is free; keywords are case-insensitive; names are best kept lower case.
- Strings in **single quotes** (double an inner quote); identifiers in backticks in MySQL; comments with `--` or `/* */`.

EXERCISE 1

Add a task, then read the list

The table `tasks` holds a to-do list (`done` is 1 for finished, 0 for open):

id	title	done
1	Buy milk	1
2	Call Ravi	0

Write a script of **two statements**: first insert a task with `id` 100, title `Review notes` and `done` 0; then return `id`, `title` and `done` for every task, ordered by `id`.

Output: the second statement's result (an `INSERT` prints nothing).

```
id | title | done
1 | Buy milk | 1
2 | Call Ravi | 0
100 | Review notes | 0
```

INPUT 1

```
CREATE TABLE tasks (
  id    INT PRIMARY KEY,
  title VARCHAR(100) NOT NULL,
  done  INT NOT NULL DEFAULT 0
);
INSERT INTO tasks VALUES
  (1, 'Buy milk', 1),
  (2, 'Call Ravi', 0);
```

OUTPUT 1

```
id | title | done
1 | Buy milk | 1
2 | Call Ravi | 0
100 | Review notes | 0
```

REFERENCE SOLUTION

```
INSERT INTO tasks (id, title, done) VALUES (100, 'Review notes', 0);

SELECT id, title, done
FROM tasks
ORDER BY id;
```

EXERCISE 2

Two queries, two results

Using the `tasks` table, write a script of **two SELECTs**: the first returns the number of tasks as `total_tasks`; the second returns every task's `title`, ordered alphabetically by `title`.

id	title	done
1	Pay rent	1
2	Call Ravi	0
3	Buy milk	0

Output: both result sets, in order, separated by a blank line.

```
total_tasks
3

title
Buy milk
Call Ravi
Pay rent
```

INPUT 1

```
CREATE TABLE tasks (
  id    INT PRIMARY KEY,
  title VARCHAR(100) NOT NULL,
  done  INT NOT NULL DEFAULT 0
);
INSERT INTO tasks VALUES
  (1, 'Pay rent', 1),
  (2, 'Call Ravi', 0),
  (3, 'Buy milk', 0);
```

OUTPUT 1

```
total_tasks
3

title
Buy milk
Call Ravi
Pay rent
```

REFERENCE SOLUTION

```sql
SELECT COUNT(*) AS total_tasks
FROM tasks;

SELECT title
FROM tasks
ORDER BY title;
```

QUIZ

1. Why is SQL called **declarative**?

A. Because every statement starts with a keyword

B. Because a query describes the result and the database chooses how to compute it

C. Because tables must be declared before use

D. Because it has no loops or variables at all

2. `CREATE`, `ALTER` and `DROP` belong to which family?

A. DML

B. DDL

C. DCL

D. TCL

3. Which is the **portable** way to write the string Pune in a condition?

A. `city = "Pune"`

B. `city = 'Pune'`

C. `city = `Pune``

D. `city = Pune`

4. How do you write the text *it's* as a SQL string literal in standard SQL?

A. `'it's'`

B. `'it''s'`

C. `"it"s"`

D. `it's`

5. Which statement about case is true in MySQL?

A. `select` and `SELECT` are different keywords

B. Keywords are case-insensitive; text comparison follows the column's collation

C. Column names are case-sensitive on every system

D. String comparison is always case-sensitive

ANSWER KEY · WHAT SQL IS — A DECLARATIVE LANGUAGE IN FIVE PARTS

1. B `SELECT name FROM students WHERE city = 'Pune'` never says whether to scan or use an index — the optimizer decides. That is the meaning of declarative. Declaring tables is DDL, a different idea.

2. B Data Definition Language changes the structure — tables, indexes, views. DML (`INSERT` / `UPDATE` / `DELETE`) changes rows, DCL (`GRANT` / `REVOKE`) permissions, TCL (`COMMIT` / `ROLLBACK`) transactions.

3. B Strings take single quotes in standard SQL and every major database. MySQL also accepts double-quoted strings by default, but PostgreSQL and SQLite read `"Pune"` as a column name; backticks quote identifiers in MySQL; a bare word is a column name.

4. B A single quote inside a single-quoted string is doubled. `'it's'` ends the string after *it*; MySQL also accepts a backslash escape (`'it\'s'`), but other databases do not.

5. B Keywords never depend on case. Column names in MySQL are case-insensitive; table names follow the file system. Whether `'pune' = 'Pune'` is true depends on the collation — MySQL's defaults are case-insensitive.

LESSON 3 · 14 MIN

MySQL and its cousins — where the dialects differ

"Which database?" is usually answered before you arrive: the company already runs MySQL, or PostgreSQL, or SQL Server. The good news is that the SQL in this course — `SELECT`, `WHERE`, joins, `GROUP BY`, subqueries, window functions — reads almost the same on all of them. The bad news is the word *almost*. This lesson introduces the major systems, says why this course teaches MySQL 8, and walks through the differences that interviewers and real migrations run into.

The family

System	Kind	Typical home
MySQL	open-source server (Oracle)	web applications, the LAMP stack, startups, many Indian product companies
MariaDB	fork of MySQL	Linux distributions, drop-in MySQL replacement
PostgreSQL	open-source server	modern web backends, analytics, GIS, anywhere strict types are valued
SQLite	embedded library, one file	phones, browsers, desktop apps, tests
SQL Server	commercial server (Microsoft)	enterprises on .NET and Windows
Oracle Database	commercial server	banks, telecoms, large enterprises

All of them are **relational** and all speak **SQL**. The servers (MySQL, PostgreSQL, SQL Server, Oracle) run as their own process that applications connect to; SQLite is a library inside the application with the database in a single file — no users, no network, no setup.

Why MySQL 8

Placement tests and interview rounds in India overwhelmingly write their SQL questions in MySQL, and most coding-practice sites judge SQL problems on it. MySQL 8 (2018 onward) added the features that modern questions need — **window functions** (`ROW_NUMBER`, `RANK`, `LAG`), **common table expressions** (`WITH`), recursive queries — so everything in this course is available there. Where PostgreSQL or SQL Server would be written differently and an interviewer might care, the lessons say so.

One note about this course itself: your exercises are judged in milliseconds by an engine that reads **MySQL syntax** and runs it on SQLite underneath. Write MySQL — `CONCAT`, `IFNULL`, `DATE_ADD(d, INTERVAL 1 DAY)`, `LIMIT` — and it is understood.

Limiting rows

The most visible difference. To take the three most expensive products:

```sql
-- MySQL, PostgreSQL, SQLite
SELECT name, price FROM products ORDER BY price DESC LIMIT 3;

-- SQL Server
SELECT TOP 3 name, price FROM products ORDER BY price DESC;

-- Standard SQL, Oracle 12c+, PostgreSQL, SQL Server 2012+ (with OFFSET)
SELECT name, price FROM products ORDER BY price DESC
FETCH FIRST 3 ROWS ONLY;
```

`LIMIT` without `ORDER BY` returns *some* three rows — whichever the engine meets first. Always pair them.

Joining strings

```sql
-- MySQL
SELECT CONCAT(first_name, ' ', last_name) AS full_name FROM employees;

-- PostgreSQL, SQLite, Oracle
SELECT first_name || ' ' || last_name AS full_name FROM employees;

-- SQL Server
SELECT first_name + ' ' + last_name AS full_name FROM employees;
```

The trap: in MySQL `||` is the **logical OR** operator (unless the server runs in `PIPES_AS_CONCAT` mode), so `'Asha' || 'Rao'` evaluates both strings as numbers and returns `0`. In MySQL, always use `CONCAT`. It returns `NULL` if any argument is `NULL`; `CONCAT_WS(' ', a, b)` ("with separator") skips NULLs.

Quoting names

```
SELECT `order`, `full name` FROM `my table`;      -- MySQL: backticks
SELECT "order", "full name" FROM "my table";      -- standard, PostgreSQL, SQL
SELECT [order], [full name] FROM [my table];      -- SQL Server (also accepts
```

Better still, never need quoting: no spaces, no keywords as names.

Auto-numbered ids

```
-- MySQL
CREATE TABLE users (id INT AUTO_INCREMENT PRIMARY KEY, email VARCHAR(100));

-- PostgreSQL (modern)
CREATE TABLE users (id INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY, email TEX
-- PostgreSQL (older): id SERIAL PRIMARY KEY

-- SQL Server
CREATE TABLE users (id INT IDENTITY(1,1) PRIMARY KEY, email NVARCHAR(100));

-- SQLite
CREATE TABLE users (id INTEGER PRIMARY KEY, email TEXT);
```

NULL helpers and conditionals

Need	MySQL	PostgreSQL	SQL Server	Oracle
first non-NULL	`COALESCE`, `IFNULL`	`COALESCE`	`COALESCE`, `ISNULL`	`COALESCE`, `NVL`
inline if	`IF(c, a, b)`, `CASE`	`CASE`	`IIF(c, a, b)`, `CASE`	`CASE`, `DECODE`

`COALESCE` and `CASE` are standard and work everywhere — prefer them when portability matters.

Dates

Date arithmetic is the least portable corner of SQL:

```
-- MySQL
SELECT DATE_ADD('2024-03-01', INTERVAL 7 DAY), DATEDIFF('2024-03-08', '2024-0
-- PostgreSQL
SELECT DATE '2024-03-01' + INTERVAL '7 days', DATE '2024-03-08' - DATE '2024-0
-- SQL Server
SELECT DATEADD(day, 7, '2024-03-01'), DATEDIFF(day, '2024-03-01', '2024-03-08
```

Note that SQL Server's `DATEDIFF` takes the unit first and the dates in the opposite order to MySQL's.

Case and comparison

MySQL's default collation (`utf8mb4_0900_ai_ci` in MySQL 8) is **case- and accent-insensitive**: `'Pune' = 'pune'` is true, and so is `'café' = 'cafe'`. PostgreSQL and SQLite compare text **case-sensitively** by default, so the same `WHERE` returns different rows. A query moved between them can quietly change its answer — normalise case with `LOWER()` when it matters.

Types and strictness

MySQL historically converted bad input instead of refusing it — `'12abc'` into an `INT` column became 12 with a warning. Since 5.7 the default **strict mode** refuses such inserts, but comparisons still convert: `WHERE id = '7'` works. PostgreSQL refuses to compare an integer

with text without a cast. SQLite goes the other way: a column's type is only a preference, and it will store text in an `INT` column. MySQL has no real boolean type — `BOOLEAN` is `TINYINT(1)`, and `TRUE` is `1` — while PostgreSQL has a true `boolean`.

What is the same

Everything the next fifteen modules spend most of their time on:

```sql
SELECT c.name, COUNT(*) AS orders, SUM(o.amount) AS spent
FROM customers AS c
JOIN orders AS o ON o.customer_id = c.id
WHERE o.amount > 0
GROUP BY c.name
HAVING COUNT(*) ⩾ 2
ORDER BY spent DESC;
```

That query runs unchanged on MySQL, PostgreSQL, SQLite, SQL Server and Oracle. Joins, grouping, subqueries, CTEs, window functions, `CASE`, `COALESCE`, `IN`, `BETWEEN`, `LIKE`, `UNION` — the core is the standard, and the standard is most of SQL.

Key takeaways

- MySQL, PostgreSQL, SQL Server and Oracle are **servers**; SQLite is an **embedded** one-file library. All speak SQL.
- This course writes **MySQL 8**: window functions and CTEs included.
- Row limits: `LIMIT` (MySQL/PostgreSQL/SQLite), `TOP` (SQL Server), `FETCH FIRST` (standard/Oracle).
- Concatenate with `CONCAT` in MySQL — `||` is OR there.
- Quote names with backticks in MySQL, double quotes in the standard; auto ids are `AUTO_INCREMENT`, `IDENTITY` or `SERIAL`.
- MySQL compares text case-insensitively by default; PostgreSQL and SQLite do not.
- Prefer standard forms (`COALESCE`, `CASE`) when the query must travel.

EXERCISE 1

Full names with CONCAT

The table `employees`:

emp_id	first_name	last_name
1	Asha	Rao
2	Vikram	Singh
3	Leela	Iyer

Return each employee's full name — first name, one space, last name — as `full_name`, ordered by `emp_id`. Use MySQL's spelling (in MySQL `||` is not concatenation). If a last name is missing, MySQL's `CONCAT` returns `NULL` — let it.

Output: one column `full_name`, by `emp_id`.

```
full_name
Asha Rao
Vikram Singh
Leela Iyer
```

INPUT 1

```
CREATE TABLE employees (
  emp_id     INT PRIMARY KEY,
  first_name VARCHAR(40) NOT NULL,
  last_name  VARCHAR(40)
);
INSERT INTO employees VALUES
  (1, 'Asha', 'Rao'),
  (2, 'Vikram', 'Singh'),
  (3, 'Leela', 'Iyer');
```

OUTPUT 1

```
full_name
Asha Rao
Vikram Singh
Leela Iyer
```

REFERENCE SOLUTION

```
SELECT CONCAT(first_name, ' ', last_name) AS full_name
FROM employees
ORDER BY emp_id;
```

EXERCISE 2

Top three with LIMIT

The table `products`:

product_id	name	price
1	Keyboard	1500
2	Monitor	9000
3	Mouse	600
4	Webcam	2500
5	Headset	2500

Return the `name` and `price` of the three most expensive products — the MySQL way. When two products cost the same, the smaller `product_id` comes first.

Output: `name | price`, at most three rows, by `price` descending then `product_id`.

```
name | price
Monitor | 9000
Webcam | 2500
Headset | 2500
```

INPUT 1

```
CREATE TABLE products (
  product_id INT PRIMARY KEY,
  name       VARCHAR(60) NOT NULL,
  price      INT NOT NULL
);
INSERT INTO products VALUES
  (1, 'Keyboard', 1500),
  (2, 'Monitor', 9000),
  (3, 'Mouse', 600),
  (4, 'Webcam', 2500),
  (5, 'Headset', 2500);
```

OUTPUT 1

```
name | price
Monitor | 9000
Webcam | 2500
Headset | 2500
```

REFERENCE SOLUTION

```
SELECT name, price
FROM products
ORDER BY price DESC, product_id
LIMIT 3;
```

QUIZ

1. In MySQL (default mode), what does `SELECT 'Asha' || 'Rao';` return?

A. `AshaRao`

B. `Asha Rao`

C. `0`

D. An error

2. Which takes the first five rows in **SQL Server**?

A. `SELECT … LIMIT 5`

B. `SELECT TOP 5 …`

C. `SELECT FIRST 5 …`

D. `SELECT … ROWNUM 5`

3. What makes **SQLite** different from MySQL and PostgreSQL?

A. It does not support SQL

B. It is an embedded library with the database in one file, no server process

C. It only runs in browsers

D. It cannot hold more than one table

4. With MySQL 8's default collation, what is `'Pune' = 'pune'`?

A. 1 (true)

B. 0 (false)

C. NULL

D. An error

5. Which NULL helper works in **every** major database?

A. `IFNULL`

B. `ISNULL`

C. `NVL`

D. `COALESCE`

ANSWER KEY · MYSQL AND ITS COUSINS — WHERE THE DIALECTS DIFFER

1. C In MySQL `||` is logical OR. Both strings convert to the number 0, and 0 OR 0 is 0. PostgreSQL, SQLite and Oracle would return `AshaRao`; in MySQL use `CONCAT`.

2. B SQL Server writes `TOP n` after `SELECT` (or `OFFSET … FETCH NEXT n ROWS ONLY`). `LIMIT` is MySQL/PostgreSQL/SQLite; the standard and Oracle 12c+ use `FETCH FIRST n ROWS ONLY`.

3. B SQLite is linked into the application and keeps the whole database in a single file — no server, users or network. It reads most of the same SQL and holds many tables.

4. A MySQL's default collations are case-insensitive, so the comparison is true. PostgreSQL and SQLite compare case-sensitively by default and would say false — one reason a query can change its answer when moved.

5. D `COALESCE` is standard SQL. `IFNULL` is MySQL/SQLite, `ISNULL` SQL Server, `NVL` Oracle.

LESSON 4 · 13 MIN

Your first SELECT — columns, tables and order

Every question you will ever ask a database starts with the same word. `SELECT` reads rows from tables and returns a result — itself a small table of rows and columns. This lesson writes the simplest useful queries: picking columns, taking all of them, computing a value with no table at all, and asking for a dependable order. Everything else in SQL is added on top of this shape.

The shape of a SELECT

```
SELECT name, city
FROM students;
```

Two clauses: `SELECT` lists **what** to return — here two columns — and `FROM` names **where** to read it. The result has exactly the columns you listed, in the order you listed them, and one row for each row of the table:

name	city
Asha	Pune
Ravi	Chennai
Meera	Pune

The columns are separated by **commas**, and a missing or extra comma is the commonest first error:

```
SELECT name city FROM students;    -- no comma: one column "name", renamed to
SELECT name, city, FROM students;  -- trailing comma: syntax error
```

The first is worse than an error: it runs. A name after an expression with no comma is read as an **alias** (the next module explains them), so the result has one column called `city` holding names.

Picking and ordering columns

You may list any of the table's columns, in any order, and the same column more than once:

```
SELECT city, name FROM students;
SELECT name, name, year FROM students;
```

The table's own column order is irrelevant to the result. A table with forty columns is routinely queried for three of them — and reading fewer columns is less work for the database and the network.

SELECT *

The star means **every column, in the table's order**:

```
SELECT * FROM students;
```

id	name	year	city
1	Asha	2	Pune
2	Ravi	1	Chennai
3	Meera	3	Pune

Use it to explore a table you do not know. Avoid it in code that other code depends on:

- it reads columns nobody uses, which costs I/O and network, and prevents some index-only plans;
- when a column is added, dropped or reordered, the result changes shape and code that reads columns by position breaks;
- it hides intent — a reviewer cannot see which data the query needs.

Interviewers like asking "what is wrong with `SELECT *`?" — those three points are the answer.

Ordering the rows

A table is a set; it has **no row order**. If you run `SELECT * FROM students` twice you will probably see the same order, because the engine happens to read the rows the same way — but nothing promises it, and after a few deletes and inserts, or with a new index, the order changes. When order matters, say so:

```sql
SELECT name, year FROM students ORDER BY name;
```

name	year
Asha	2
Meera	3
Ravi	1

`ORDER BY` comes **after** `FROM` (and after `WHERE`, once there is one). It sorts ascending by default — numbers small to large, text A to Z, dates old to new; add `DESC` for descending:

```sql
SELECT name, year FROM students ORDER BY year DESC;
```

You may order by a column you did not select, and by several columns — the second breaks ties in the first:

```sql
SELECT name FROM students ORDER BY city, name;
```

Module 3 covers sorting in depth. For now, one rule that every exercise in the course relies on: **if the output is printed or compared, the query ends in an `ORDER BY` that fixes the order completely** — ordering by the primary key is the simplest way to guarantee that.

SELECT without a table

`SELECT` can compute values from nothing:

```sql
SELECT 2 + 3 AS total, 'hello' AS greeting;
```

total	greeting
5	hello

One row, computed from literals. It is the quickest way to try out an expression or a function. MySQL, PostgreSQL, SQLite and SQL Server accept it as written; Oracle needs a table and supplies the one-row dummy table `DUAL` (`SELECT 2 + 3 FROM dual`), which MySQL also accepts for compatibility.

Mixing columns and values

A select list can hold any mix of columns, literals and expressions:

```sql
SELECT name, 'student' AS role, year + 1 AS next_year
FROM students
ORDER BY id;
```

name	role	next_year
Asha	student	3
Ravi	student	2
Meera	student	4

The literal `'student'` is repeated on every row; `year + 1` is computed per row. The stored data is untouched — `SELECT` only reads. Expressions get a lesson of their own in the next module.

Qualified names

A column can be written with its table in front:

```sql
SELECT students.name, students.city FROM students;
```

With one table it changes nothing. With two tables in a join, both of which might have a `name` column, the prefix is how you say which one you mean — and a short **table alias** (`FROM students AS s`, then `s.name`) keeps it readable. You will meet that in the joins module.

Formatting habits

Write a clause per line, keywords in capitals, a column per line once there are more than three or four:

```
SELECT id,
       name,
       city
FROM   students
ORDER  BY id;
```

The database does not care. Your colleagues, your future self and your interviewer do, and the clause-per-line layout makes the next lesson's subject — the order in which clauses run — easy to see.

Reading an error

Typical first-week errors and what they mean:

- `Unknown column 'nmae' in 'field list'` — a typo in a column name, or a column of another table.
- `Table 'school.student' doesn't exist` — the table name is wrong (`students`, plural).
- `You have an error in your SQL syntax … near 'FROM students'` — usually a trailing comma or a missing one just before the quoted text.

The quoted fragment is where the parser gave up; the mistake is usually just **before** it.

Key takeaways

- `SELECT columns FROM table` returns those columns for every row; the result is a table.
- Separate columns with commas; a missing comma silently turns the next name into an alias.
- `SELECT *` is for exploring; list columns in code.
- Rows have **no order** without `ORDER BY`; `DESC` reverses; a second column breaks ties.
- `SELECT` can compute from literals with no table; Oracle needs `FROM dual`.
- Literals and expressions can sit beside columns; `SELECT` never changes the data.

EXERCISE 1

Choose the columns

The table `customers`:

customer_id	name	email	city	joined
1	Asha	asha@example.com	Pune	2023-04-01
2	Ravi	ravi@example.com	Chennai	2023-06-15
3	Meera	NULL	Pune	2024-01-09

Return only each customer's `name` and `city`, in that order, ordered by `customer_id`.

Output: `name | city`, by `customer_id`.

```
name | city
Asha | Pune
Ravi | Chennai
Meera | Pune
```

INPUT 1

```
CREATE TABLE customers (
  customer_id INT PRIMARY KEY,
  name        VARCHAR(60) NOT NULL,
  email       VARCHAR(100),
  city        VARCHAR(40),
  joined      DATE
);
INSERT INTO customers VALUES
  (1, 'Asha', 'asha@example.com',
'Pune', '2023-04-01'),
  (2, 'Ravi', 'ravi@example.com',
'Chennai', '2023-06-15'),
  (3, 'Meera', NULL, 'Pune', '2024-01-
09');
```

OUTPUT 1

```
name | city
Asha | Pune
Ravi | Chennai
Meera | Pune
```

REFERENCE SOLUTION

```
SELECT name, city
FROM customers
ORDER BY customer_id;
```

EXERCISE 2

Your column order, plus a label

Using `customers`, return three columns in this order: `city`, `name`, and a column `kind` that is the text `customer` on every row. Order the rows by `name`; two customers can share a name, so break ties by `customer_id`.

customer_id	name	email	city	joined
1	Ravi	ravi@example.com	Chennai	2023-06-15
2	Asha	asha@example.com	Pune	2023-04-01
3	Meera	NULL	Pune	2024-01-09

Output: `city | name | kind`, by `name` then `customer_id`.

```
city | name | kind
Pune | Asha | customer
Pune | Meera | customer
Chennai | Ravi | customer
```

INPUT 1

```
CREATE TABLE customers (
  customer_id INT PRIMARY KEY,
  name        VARCHAR(60) NOT NULL,
  email       VARCHAR(100),
  city        VARCHAR(40),
  joined      DATE
);
INSERT INTO customers VALUES
  (1, 'Ravi', 'ravi@example.com',
'Chennai', '2023-06-15'),
  (2, 'Asha', 'asha@example.com',
'Pune', '2023-04-01'),
  (3, 'Meera', NULL, 'Pune', '2024-01-
09');
```

OUTPUT 1

```
city | name | kind
Pune | Asha | customer
Pune | Meera | customer
Chennai | Ravi | customer
```

REFERENCE SOLUTION

```sql
SELECT city, name, 'customer' AS kind
FROM customers
ORDER BY name, customer_id;
```

QUIZ

1. What does `SELECT name city FROM students;` return?

A. Two columns, name and city

B. A syntax error

C. One column called `city` holding the names

D. One column called `name` holding the cities

2. Which is **not** a reason to avoid `SELECT *` in application code?

A. It reads columns nobody uses

B. The result changes shape when the table's columns change

C. It hides which data the query needs

D. It returns the rows in a random order

3. What does `SELECT 2 + 3 AS total;` return in MySQL?

A. An error: no table

B. One row with total = 5

C. Five rows

D. The string `2 + 3`

4. Can a query order by a column it does not select, as in `SELECT name FROM students ORDER BY year`?

A. No, ORDER BY may only use selected columns

B. Yes

C. Only with `DESC`

D. Only if `year` is the primary key

ANSWER KEY · YOUR FIRST SELECT — COLUMNS, TABLES AND ORDER

1. C Without a comma, `city` is read as an alias for `name`. The query runs and returns names under the heading `city` — a silent bug, worse than an error.

2. D Row order has nothing to do with `*` — without `ORDER BY` every query's order is unspecified. The other three are the real costs of `SELECT *`.

3. B A `SELECT` without `FROM` computes one row from its expressions. Oracle needs `FROM dual`; MySQL accepts both forms.

4. B The rows still carry every column when they are sorted; the select list only decides what is shown. The exception is after `DISTINCT` or `GROUP BY`.

LESSON 5 · 14 MIN

How a query runs — the logical order of the clauses

You write a query in the order SQL's grammar demands — `SELECT` first, then `FROM`, then the rest. The database does not evaluate it in that order. Knowing the order it does use, the **logical order of execution**, is one of the most useful things in SQL: it explains a whole class of errors ("unknown column" for an alias you just defined), it tells you where each condition belongs, and it is a favourite interview question. This lesson also follows a query through the database — parser, optimizer, executor — so that "the database decides how" stops being magic.

Written order and logical order

The clauses of a full `SELECT` are written in this order:

```
SELECT   DISTINCT columns, expressions
FROM     tables and joins
WHERE    row conditions
GROUP BY grouping columns
HAVING   group conditions
ORDER BY sort keys
LIMIT    count;
```

They are evaluated, logically, in this one:

Step	Clause	What it does
1	`FROM` / `JOIN`	assemble the input rows from the tables
2	`WHERE`	keep the rows whose condition is true
3	`GROUP BY`	collapse the rows into groups
4	`HAVING`	keep the groups whose condition is true
5	`SELECT`	compute the output columns and their aliases
6	`DISTINCT`	drop duplicate output rows
7	`ORDER BY`	sort the output
8	`LIMIT` / `OFFSET`	keep a slice of the sorted rows

Read it as a pipeline: each step takes the previous step's rows and hands its own on. `GROUP BY` and `HAVING` are the subject of the aggregation module; the rest you can use already.

Consequence 1: aliases do not exist in WHERE

```sql
SELECT name, price * qty AS line_total
FROM order_items
WHERE line_total > 100;          -- error: Unknown column 'line_total' in 'whe
```

`WHERE` is step 2; the alias `line_total` is created in step 5. When the rows are being filtered there is no such column. Repeat the expression instead:

```sql
SELECT name, price * qty AS line_total
FROM order_items
WHERE price * qty > 100;
```

The database computes the product once per row either way; repeating it costs nothing. (Later modules show the other fix — compute the alias in a subquery or a CTE, then filter on it outside.)

Consequence 2: aliases do exist in ORDER BY

```sql
SELECT name, price * qty AS line_total
FROM order_items
WHERE price * qty > 100
ORDER BY line_total DESC;
```

`ORDER BY` is step 7, after `SELECT` has named its columns, so the alias is visible. This holds in every major database. (MySQL is also lenient enough to accept select aliases in `GROUP BY` and `HAVING`; PostgreSQL accepts them in `GROUP BY` but not `HAVING`; SQL Server accepts neither. Portable code repeats the expression there too.)

Consequence 3: ORDER BY can see columns you did not select

`ORDER BY` runs after `SELECT`, yet it may sort by a column that is not in the select list:

```sql
SELECT name FROM students ORDER BY year DESC, name;
```

That works because, logically, the rows still carry all their columns until the very end — the select list decides what is *shown*. The exception: after `DISTINCT` or `GROUP BY` the original rows have been merged, so `ORDER BY` may only use what survived.

Consequence 4: LIMIT is last

`LIMIT 3` takes the first three rows **of the sorted result**. It does not take three rows and then sort them. That is why `ORDER BY … LIMIT` answers "top three", and why `LIMIT` without `ORDER BY` answers "any three".

Consequence 5: WHERE sees rows, not results

`WHERE` decides about one input row at a time, before anything is computed across rows. A condition such as "the price is above the average price" cannot be written as `WHERE price > AVG(price)` — at step 2 there is no average yet. Aggregates belong to `HAVING` (about groups) or to a subquery. You will meet both; the error message `Invalid use of group function` means exactly this.

A worked trace

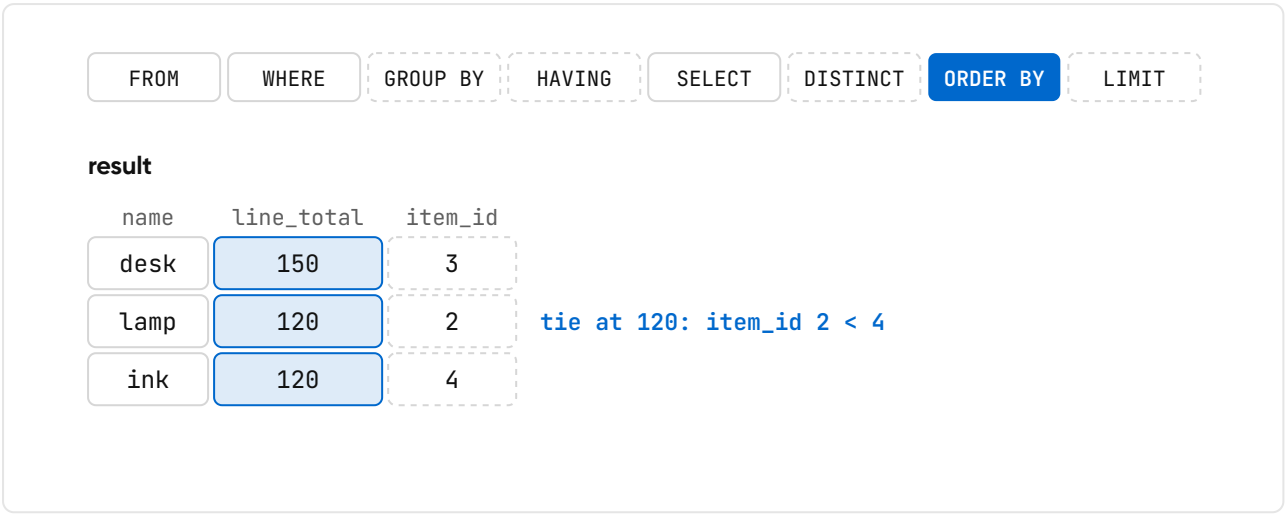
```sql
SELECT name, price * qty AS line_total
FROM order_items
WHERE price * qty >= 100
ORDER BY line_total DESC, item_id;
```

with

item_id	name	price	qty
1	pen	10	5
2	lamp	60	2
3	desk	150	1
4	ink	40	3

1. `FROM` — the four rows.
2. `WHERE price * qty >= 100` — pen gives 50, dropped; lamp 120, desk 150, ink 120 kept.
3. `SELECT` — each kept row becomes `(name, line_total)`, but still remembers its `item_id`.
4. `ORDER BY line_total DESC, item_id` — desk 150; then lamp and ink tie on 120, and `item_id` puts lamp (2) before ink (4).

The logical order: each step hands its rows to the next



1. FROM runs first and produces the input: all 4 rows of order_items. The query has no GROUP BY, HAVING, DISTINCT or LIMIT, so those steps pass the rows straight through.
2. WHERE tests each row on its own. The alias line_total does not exist yet, so the condition repeats price * qty: pen gives 50 and is dropped; the other 3 rows move on.
3. SELECT computes the output columns and names line_total. The rows still remember item_id even though the select list does not show it.
4. ORDER BY runs after SELECT, so it can use the alias: desk's 150 first, then lamp and ink tied at 120, which item_id puts in order. The hidden column is dropped from what is printed.

What the database actually does

The logical order defines **what the answer is**. How the engine computes it is a separate matter, decided per query:

1. **Parser.** The text is checked against the grammar and turned into a tree. Syntax errors come from here.
2. **Resolver.** Names are looked up: does `order_items` exist, does it have a `qty` column, are the types compatible? "Unknown column" errors come from here.
3. **Optimizer.** The tree is turned into an **execution plan**. The optimizer considers alternatives — scan the table or use an index, which table to read first in a join, hash join or nested loop, sort now or read an index that is already sorted — and estimates the cost of each from **statistics** about the data (row counts, how many distinct values a column has). It picks the cheapest.
4. **Executor.** The plan runs, reading pages from memory or disk, and rows stream back to the client.

The plan is allowed to do things in any order as long as the result is the one the logical order defines. With an index on `price`, `ORDER BY price LIMIT 5` can read five index entries and

stop — no sort at all, and the `LIMIT` effectively runs first. A condition in `WHERE` on an indexed column can become an index lookup instead of a filter over every row.

You can ask to see the plan:

```
EXPLAIN SELECT name FROM students WHERE city = 'Pune';
```

MySQL answers with a row per table: which index it would use (`key`), how many rows it expects to examine (`rows`), and notes such as `Using where` or `Using filesort`. Reading plans is the subject of the indexes module; for now it is enough to know they exist and that the optimizer, not the order you wrote the clauses in, decides the work.

Where each condition belongs

The logical order also settles a design question: put a row condition in `WHERE`, as early as possible. Rows removed at step 2 are never grouped, computed, sorted or sent. Filtering is the cheapest thing a query does, and the optimizer is best at it.

Key takeaways

- Written order starts with `SELECT`; the logical order is **FROM → WHERE → GROUP BY → HAVING → SELECT → DISTINCT → ORDER BY → LIMIT**.
- A select **alias cannot be used in WHERE** (it does not exist yet) but **can be used in ORDER BY**.
- `ORDER BY` may sort by unselected columns, except after `DISTINCT` / `GROUP BY`.
- `LIMIT` slices the sorted result; `WHERE` cannot use aggregates.
- Parser → resolver → optimizer → executor; the optimizer picks a plan from statistics, and `EXPLAIN` shows it.
- The plan may reorder work freely as long as the result matches the logical order.

EXERCISE 1

Filter on an expression, sort by its alias

The table `order_items`:

item_id	name	price	qty
1	pen	10	5
2	lamp	60	2
3	desk	150	1
4	ink	40	3

Return `name` and the line total `price * qty` as `line_total`, only for lines whose total is **at least 100**, ordered by `line_total` descending, ties by `item_id`. Remember which clauses can see the alias.

Output: `name | line_total`.

```
name | line_total
desk | 150
lamp | 120
ink | 120
```

INPUT 1

```
CREATE TABLE order_items (
  item_id INT PRIMARY KEY,
  name    VARCHAR(60) NOT NULL,
  price   INT NOT NULL,
  qty     INT NOT NULL
);
INSERT INTO order_items VALUES
  (1, 'pen', 10, 5),
  (2, 'lamp', 60, 2),
  (3, 'desk', 150, 1),
  (4, 'ink', 40, 3);
```

OUTPUT 1

```
name | line_total
desk | 150
lamp | 120
ink | 120
```

REFERENCE SOLUTION

```
SELECT name, price * qty AS line_total
FROM order_items
WHERE price * qty ⩾ 100
ORDER BY line_total DESC, item_id;
```

EXERCISE 2

Sort by a column you do not show

The table `students`:

id	name	score
1	Asha	72
2	Ravi	91
3	Meera	85

Return only the students' `name`, ordered by `score` from highest to lowest; students with equal scores in alphabetical order of `name`. A missing score (NULL) counts as the lowest, which is where MySQL puts NULLs in a descending sort.

Output: one column `name`.

```
name
Ravi
Meera
Asha
```

INPUT 1

```
CREATE TABLE students (
  id    INT PRIMARY KEY,
  name  VARCHAR(60) NOT NULL,
  score INT
);
INSERT INTO students VALUES
  (1, 'Asha', 72),
  (2, 'Ravi', 91),
  (3, 'Meera', 85);
```

OUTPUT 1

```
name
Ravi
Meera
Asha
```

REFERENCE SOLUTION

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

QUIZ

1. What is the **logical** order of these clauses?

A. SELECT, FROM, WHERE, ORDER BY

B. FROM, WHERE, SELECT, ORDER BY

C. FROM, SELECT, WHERE, ORDER BY

D. WHERE, FROM, SELECT, ORDER BY

2. Why does `SELECT price * qty AS total FROM items WHERE total > 100;` fail?

A. `total` is a reserved word

B. WHERE is evaluated before SELECT, so the alias does not exist yet

C. Aliases need quotes

D. Arithmetic is not allowed in SELECT

3. Given `SELECT name FROM t ORDER BY score DESC LIMIT 2`, what happens first?

A. Two rows are taken, then sorted

B. The rows are sorted, then the first two are kept

C. LIMIT and ORDER BY run in the order written

D. It depends on the table's insertion order

4. Which component chooses whether to use an index for a query?

A. The parser

B. The optimizer

C. The client

D. The WHERE clause

5. Why can't `WHERE price > AVG(price)` work?

A. AVG needs a GROUP BY

B. WHERE decides about one row at a time, before any aggregate exists

C. AVG cannot be compared with a column

D. It works in MySQL

ANSWER KEY · HOW A QUERY RUNS — THE LOGICAL ORDER OF THE CLAUSES

1. B Rows are assembled (FROM), filtered (WHERE), shaped (SELECT), then sorted (ORDER BY). The written order starts with SELECT, which is why the two are easy to confuse.

2. B The alias is created at the SELECT step, after WHERE has filtered. Repeat the expression in WHERE, or filter on the alias in an outer query.

3. B LIMIT is the last logical step, so it slices the sorted result — which is why ORDER BY + LIMIT means top-N. (A plan may stop early using an index, but the answer is the same.)

4. B The parser checks syntax, the resolver checks names, and the optimizer builds the execution plan — index or scan, join order, sort or not — from statistics. `EXPLAIN` shows its choice.

5. B WHERE runs at step 2, row by row; aggregates are computed later. MySQL answers *Invalid use of group function*. Compare with a subquery instead, or use HAVING for conditions on groups.

CHECKPOINT · 25 MIN

Checkpoint — Foundations

This checkpoint covers the whole module: what a relational database is — tables, typed columns, rows without order, primary and foreign keys — and what the DBMS does; SQL as a declarative language and its statement families; MySQL beside PostgreSQL, SQLite and SQL Server; the basic `SELECT`; and the logical order in which a query's clauses are evaluated.

How it works. Twelve questions and three queries. You need 70% on the questions and every query accepted to clear the module. You can retake it as often as you like; your best score counts.

Before you start, make sure you can answer these from memory:

- What are a row, a column and a table's schema, and why does a table have no row order?
- What two properties must a primary key have, and what does a foreign key point at?
- Name the five statement families and one statement from each.
- Why does `SELECT name city FROM students` run without an error, and what does it return?
- How do MySQL, SQL Server and the SQL standard each take the first three rows?
- How do you join two strings in MySQL, and what does `||` do there?
- In what order are `FROM`, `WHERE`, `SELECT`, `ORDER BY` and `LIMIT` evaluated, and what follows for aliases?
- What is wrong with `SELECT *` in application code?

The three queries are a whole table read back in a fixed order, a list of chosen columns with a literal and a computed column beside them, and a filtered list ordered by a computed alias with a tie-breaker.

EXERCISE 1

Read the catalogue in order

The table `courses`:

course_id	code	title	credits
30	CS301	Databases	4
10	CS101	Programming	3
20	CS201	Data Structures	4

Return every column of every course, ordered by `course_id`.

Output: `course_id | code | title | credits`.

```
course_id | code | title | credits
10 | CS101 | Programming | 3
20 | CS201 | Data Structures | 4
30 | CS301 | Databases | 4
```

INPUT 1

```
CREATE TABLE courses (
  course_id INT PRIMARY KEY,
  code      VARCHAR(10) NOT NULL,
  title     VARCHAR(80) NOT NULL,
  credits   INT NOT NULL
);
INSERT INTO courses VALUES
  (30, 'CS301', 'Databases', 4),
  (10, 'CS101', 'Programming', 3),
  (20, 'CS201', 'Data Structures', 4);
```

OUTPUT 1

```
course_id | code | title | credits
10 | CS101 | Programming | 3
20 | CS201 | Data Structures | 4
30 | CS301 | Databases | 4
```

REFERENCE SOLUTION

```sql
SELECT *
FROM courses
ORDER BY course_id;
```

EXERCISE 2

A report row per student

The table `students`:

student_id	name	branch	credits
2	Ravi	ECE	18
1	Asha	CSE	20

Return, in this order: `student_id`, `name`, the contact hours `credits * 15` as `contact_hours`, and the text `active` as `status`. Order by `student_id`.

Output: `student_id | name | contact_hours | status`.

```
student_id | name | contact_hours | status
1 | Asha | 300 | active
2 | Ravi | 270 | active
```

INPUT 1

```
CREATE TABLE students (
  student_id INT PRIMARY KEY,
  name       VARCHAR(60) NOT NULL,
  branch     VARCHAR(10),
  credits    INT NOT NULL
);
INSERT INTO students VALUES
  (2, 'Ravi', 'ECE', 18),
  (1, 'Asha', 'CSE', 20);
```

OUTPUT 1

```
student_id | name | contact_hours |
status
1 | Asha | 300 | active
2 | Ravi | 270 | active
```

REFERENCE SOLUTION

```sql
SELECT student_id,
       name,
       credits * 15 AS contact_hours,
       'active' AS status
FROM students
ORDER BY student_id;
```

EXERCISE 3

Big orders first

The table `orders`:

order_id	customer	amount	discount
1	Asha	1200	200
2	Ravi	450	0
3	Meera	800	100
4	Dev	1100	100

Return `order_id`, `customer` and the net amount `amount - discount` as `net`, only for orders whose net is **greater than 500**, ordered by `net` descending, ties by `order_id`.

Output: `order_id | customer | net`.

```
order_id | customer | net
1 | Asha | 1000
4 | Dev | 1000
3 | Meera | 700
```

INPUT 1

```
CREATE TABLE orders (
  order_id INT PRIMARY KEY,
  customer VARCHAR(60) NOT NULL,
  amount   INT NOT NULL,
  discount INT NOT NULL DEFAULT 0
);
INSERT INTO orders VALUES
  (1, 'Asha', 1200, 200),
  (2, 'Ravi', 450, 0),
  (3, 'Meera', 800, 100),
  (4, 'Dev', 1100, 100);
```

OUTPUT 1

```
order_id | customer | net
1 | Asha | 1000
4 | Dev | 1000
3 | Meera | 700
```

REFERENCE SOLUTION

```sql
SELECT order_id, customer, amount - discount AS net
FROM orders
WHERE amount - discount > 500
ORDER BY net DESC, order_id;
```

CHECKPOINT QUESTIONS

1. Which statement about a relational table is true?

A. Its rows are stored and returned in insertion order

B. Each column has a name and a type, and the rows form a set with no order

C. Each row may have a different set of columns

D. Columns are addressed by number

2. Why is a surrogate integer `id` usually preferred as a primary key over an email address?

A. Integers are always faster to type

B. An email can change, and a key that changes must change everywhere it is referenced

C. Emails cannot be unique

D. MySQL does not allow text keys

3. Which family does `GRANT SELECT ON orders TO analyst;` belong to?

A. DDL

B. DML

C. DCL

D. TCL

4. Which is MySQL's way to name a column called `order` (a keyword)?

A. `'order'`

B. `` `order` ``

C. `[order]`

D. `(order)`

5. What does this return?

```
SELECT 'SQL' AS lang, 10 + 5 AS n;
```

A. An error: no FROM

B. `lang | n` then `SQL | 15`

C. `lang | n` then `SQL | 10 + 5`

D. Nothing

6. A colleague writes `SELECT title, year, FROM books;`. What happens?

A. Two columns are returned

B. A syntax error near FROM

C. An empty result

D. A third, empty column

7. Which spelling joins first and last names in **MySQL**?

A. `first_name || ' ' || last_name`

B. `first_name + ' ' + last_name`

C. `CONCAT(first_name, ' ', last_name)`

D. `JOIN(first_name, last_name)`

8. Which query takes the three cheapest products in the **SQL standard** form?

A. `SELECT TOP 3 * FROM products ORDER BY price`

B. `SELECT * FROM products ORDER BY price FETCH FIRST 3 ROWS ONLY`

C. `SELECT * FROM products LIMIT 3`

D. `SELECT * FROM products WHERE ROWNUM ⩽ 3`

9. In which clause can a select-list alias be used in every major database?

A. WHERE

B. ORDER BY

C. FROM

D. JOIN … ON

10. What does this return for rows (name, score) = (Asha, 70), (Ravi, 90), (Meera, 70)?

```
SELECT name FROM students ORDER BY score DESC, name LIMIT 2;
```

A. Ravi, Asha

B. Ravi, Meera

C. Asha, Meera

D. Ravi only

11. What does the query **optimizer** produce?

A. The parse tree

B. An execution plan

C. The result rows

D. The table's statistics

12. Which is the best description of a **foreign key**?

A. A key imported from another database

B. A column whose values must match a primary key in another table

C. A second primary key

D. An index on a text column

ANSWER KEY · CHECKPOINT — FOUNDATIONS

1. B A table has a fixed list of typed, named columns, and its rows are a set — no inherent order. Order comes only from `ORDER BY`.

2. B Foreign keys copy the primary key's value. If that value changes, every reference must change too. A meaningless id never needs to. Text keys are allowed, and emails can be unique.

3. C Data Control Language manages permissions: `GRANT` and `REVOKE`.

4. B MySQL quotes identifiers with backticks. Single quotes make a string; square brackets are SQL Server; the standard uses double quotes.

5. B A SELECT without FROM computes one row from its literals and expressions; `10 + 5` is evaluated.

6. B The trailing comma after `year` leaves the parser expecting another column, so it fails at `FROM`. The mistake is just before the quoted fragment.

7. C `CONCAT` is MySQL's concatenation. `||` is OR in MySQL (concatenation in PostgreSQL/SQLite/Oracle); `+` is SQL Server's.

8. B `FETCH FIRST n ROWS ONLY` is the standard (Oracle 12c+, PostgreSQL, SQL Server with OFFSET). `TOP` is SQL Server, `LIMIT` MySQL/PostgreSQL/SQLite — and the `LIMIT` option has no ORDER BY, so it is not the cheapest three anyway.

9. B ORDER BY is evaluated after SELECT. WHERE, FROM and ON run before the alias exists. (MySQL also accepts aliases in GROUP BY and HAVING; that is not portable.)

10. A Sorted by score descending: Ravi (90), then the 70s in name order — Asha, Meera. LIMIT 2 keeps Ravi and Asha.

11. B The optimizer chooses how to run the query — indexes, join order, algorithms — and outputs a plan, using statistics as its input. The executor then runs the plan to produce rows.

12. B A foreign key references another table's primary key, recording a relationship and letting the database refuse references to rows that do not exist.

CodeKairo · SQL study plan · Module 1: Foundations. Solve the exercises online to track your progress and earn XP.