SkillNewton

SQL Query Examples by Task

Practical SQL query examples on one small schema: filtering, pagination, joins, aggregates, subqueries, CTEs, window functions, upserts and dates.

> These SQL query examples are organised by task and all run against one small schema of students, courses and enrolments, defined in the first section. Each row shows the full query and what it returns, so you can copy the pattern into your own project. They target MySQL 8; window functions and CTEs also work in MariaDB 10.2 and later. Use them to practise after the basics, or as a reference when you need a ranking, running total or upsert.

How to use it

1. **Create the example schema.** Run the CREATE TABLE statements from the first section in a practice database and insert a few rows of your own.
2. **Find your task.** Pick the section that matches what you need, such as pagination, ranking or finding missing rows.
3. **Run and adapt.** Run the example, compare the result with the description, then swap in your own table and column names.
4. **Make changes safely.** For UPDATE and DELETE, run the matching SELECT first and wrap the change in a transaction.

Example schema

`students (id, name, email, city, joined_on)`	One row per learner. `email` is unique.
`courses (id, title, fee, duration_weeks)`	One row per course. `fee` is in rupees.
`enrolments (id, student_id, course_id, enrolled_on, status, score)`	Links students to courses; `score` is NULL until assessed.
`UNIQUE KEY uq_student_course (student_id, course_id)`	One enrolment per student per course. The upsert examples rely on this key.
`status ENUM('active', 'completed', 'dropped')`	Where each enrolment stands.

```
CREATE TABLE students (
  id        INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  name      VARCHAR(100) NOT NULL,
  email     VARCHAR(255) NOT NULL UNIQUE,
  city      VARCHAR(60),
  joined_on DATE NOT NULL
);

CREATE TABLE courses (
  id             INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  title          VARCHAR(150) NOT NULL,
  fee            DECIMAL(10,2) NOT NULL,
  duration_weeks TINYINT UNSIGNED NOT NULL
);

CREATE TABLE enrolments (
  id          INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
```

```
  student_id  INT UNSIGNED NOT NULL,
  course_id   INT UNSIGNED NOT NULL,
  enrolled_on DATE NOT NULL,
  status      ENUM('active','completed','dropped') NOT NULL DEFAULT 'active',
  score       TINYINT UNSIGNED NULL,
  UNIQUE KEY uq_student_course (student_id, course_id),
  FOREIGN KEY (student_id) REFERENCES students(id),
  FOREIGN KEY (course_id)  REFERENCES courses(id)
);

INSERT INTO students (name, email, city, joined_on) VALUES
  ('Asha', 'asha@example.com', 'Pune', '2026-01-05'),
  ('Ravi', 'ravi@example.com', 'Delhi', '2026-02-10'),
  ('Meera', 'meera@example.com', NULL, '2026-03-15');

INSERT INTO courses (title, fee, duration_weeks) VALUES
  ('SEO Foundation', 15000.00, 8),
  ('PHP Foundation', 18000.00, 12);

INSERT INTO enrolments (student_id, course_id, enrolled_on, status, score) VALUES
  (1, 1, '2026-01-06', 'completed', 82),
  (1, 2, '2026-03-01', 'active', NULL),
  (2, 1, '2026-02-11', 'completed', 91);
```

Selecting and filtering

`SELECT name, email FROM students WHERE city = 'Pune';`	Name and email of every student in Pune.
`SELECT name FROM students WHERE city IN ('Pune', 'Mumbai', 'Delhi');`	Students in any of the three cities.
`SELECT name FROM students WHERE city IS NULL;`	Students with no city recorded (`= NULL` would return nothing).
`SELECT title, fee FROM courses WHERE fee BETWEEN 10000 AND 20000;`	Courses costing from 10,000 to 20,000 rupees inclusive.
`SELECT name FROM students WHERE email LIKE '%@example.com';`	Students whose email ends with that domain.
`SELECT name, COALESCE(city, 'Not given') AS city FROM students;`	Every student, showing "Not given" instead of NULL.

`SELECT title,`
`CASE`
`WHEN duration_weeks <= 4 THEN 'Short'`
`WHEN duration_weeks <= 12 THEN 'Standard'`
`ELSE 'Long'`
`END AS length_band`
`FROM courses;`
	Each course with a label worked out from its duration.

Sorting and pagination

`SELECT title, fee FROM courses ORDER BY fee DESC, title;`	Most expensive first; equal fees sorted by title.
`SELECT name, joined_on FROM students ORDER BY joined_on DESC LIMIT 5;`	The five most recent students.

`SELECT id, name`
`FROM students`
`ORDER BY id`
`LIMIT 20 OFFSET 40;`
	Page 3 at 20 per page (rows 41 to 60). Simple, but slows down on very deep pages.

`SELECT id, name`
`FROM students`
`WHERE id > 120`
`ORDER BY id`
`LIMIT 20;`
	Keyset pagination: the 20 students after the last id shown (120). Stays fast on large tables.

`SELECT COUNT(*) FROM students;`	Total rows, for working out the number of pages.
`SELECT name, city FROM students ORDER BY city IS NULL, city;`	Sorted by city with NULL cities last (`city IS NULL` is 0 or 1).

Joins

`SELECT s.name, c.title, e.status`
`FROM enrolments e`
`JOIN students s ON s.id = e.student_id`
`JOIN courses c ON c.id = e.course_id;`
	One row per enrolment with the student name and course title (INNER JOIN).

`SELECT s.name, c.title`
`FROM students s`
`LEFT JOIN enrolments e ON e.student_id = s.id`
`LEFT JOIN courses c ON c.id = e.course_id;`
	Every student, including those with no enrolment (their title is NULL).

`SELECT s.name`
`FROM students s`
`LEFT JOIN enrolments e ON e.student_id = s.id`
`WHERE e.id IS NULL;`
	Students who have never enrolled.

`SELECT c.title, s.name`
`FROM courses c`
`LEFT JOIN enrolments e`
`ON e.course_id = c.id AND e.status = 'active'`
`LEFT JOIN students s ON s.id = e.student_id;`
	Every course with its active students. The status test sits in ON, so courses with no active students still appear.

`SELECT s.name, s.email`
`FROM students s`
`JOIN enrolments e ON e.student_id = s.id`
`JOIN courses c ON c.id = e.course_id`
`WHERE c.title = 'PHP Foundation';`
	Name and email of everyone enrolled on one course.

Aggregates and GROUP BY

`SELECT c.title, COUNT(e.id) AS enrolments`
`FROM courses c`
`LEFT JOIN enrolments e ON e.course_id = c.id`
`GROUP BY c.id, c.title`
`ORDER BY enrolments DESC;`
	Enrolments per course, including courses with zero (COUNT of a column ignores NULLs).

`SELECT status, COUNT(*) AS total`
`FROM enrolments`
`GROUP BY status;`
	How many enrolments are active, completed and dropped.

SELECT c.title, ROUND(AVG(e.score), 1) AS avg_score FROM enrolments e JOIN courses c ON c.id = e.course_id WHERE e.score IS NOT NULL GROUP BY c.id, c.title;	Average assessed score per course, to one decimal place.
SELECT s.name, SUM(c.fee) AS fees FROM enrolments e JOIN students s ON s.id = e.student_id JOIN courses c ON c.id = e.course_id GROUP BY s.id, s.name HAVING SUM(c.fee) > 20000;	Students whose total course fees exceed 20,000 rupees. HAVING filters after grouping.
SELECT city, COUNT(*) AS students FROM students WHERE city IS NOT NULL GROUP BY city ORDER BY students DESC LIMIT 3;	The three cities with the most students.
SELECT SUM(status = 'completed') AS completed, COUNT(*) AS total FROM enrolments;	Completed and total enrolments in one row (in MySQL a true comparison counts as 1).

Subqueries and CTEs (WITH)

SELECT title, fee FROM courses WHERE fee > (SELECT AVG(fee) FROM courses);	Courses priced above the average fee.
SELECT name FROM students WHERE id IN (SELECT student_id FROM enrolments WHERE status = 'completed');	Students who have completed at least one course.
SELECT s.name FROM students s WHERE NOT EXISTS (SELECT 1 FROM enrolments e WHERE e.student_id = s.id);	Students with no enrolments. NOT EXISTS is safe with NULLs, unlike NOT IN.
WITH course_stats AS (SELECT course_id, COUNT(*) AS learners, AVG(score) AS avg_score FROM enrolments GROUP BY course_id) SELECT c.title, cs.learners, ROUND(cs.avg_score, 1) AS avg_score FROM courses c JOIN course_stats cs ON cs.course_id = c.id;	A named, readable intermediate result (CTE) joined back to courses: learners and average score per course.

`WITH RECURSIVE weeks AS (`
`SELECT 1 AS week`
`UNION ALL`
`SELECT week + 1 FROM weeks WHERE week < 12`
`)`
`SELECT week FROM weeks;`
	Numbers 1 to 12 from a recursive CTE, useful for filling gaps in reports.

Window functions

`SELECT s.name, c.title, e.score,`
`RANK() OVER (PARTITION BY e.course_id ORDER BY e.score DESC) AS course_rank`
`FROM enrolments e`
`JOIN students s ON s.id = e.student_id`
`JOIN courses c ON c.id = e.course_id`
`WHERE e.score IS NOT NULL;`
	Every scored enrolment with its rank inside the course. Ties share a rank and leave a gap (1, 1, 3); DENSE_RANK gives 1, 1, 2.

`WITH ranked AS (`
`SELECT e.course_id, s.name, e.score,`
`ROW_NUMBER() OVER (PARTITION BY e.course_id ORDER BY e.score DESC) AS rn`
`FROM enrolments e`
`JOIN students s ON s.id = e.student_id`
`WHERE e.score IS NOT NULL`
`)`
`SELECT course_id, name, score`
`FROM ranked`
`WHERE rn = 1;`
	The single top scorer per course. ROW_NUMBER never ties, so exactly one row per course.

`SELECT enrolled_on,`
`COUNT(*) AS new_enrolments,`
`SUM(COUNT(*)) OVER (ORDER BY enrolled_on) AS running_total`
`FROM enrolments`
`GROUP BY enrolled_on`
`ORDER BY enrolled_on;`
	Enrolments per day plus a running total to date.

`SELECT student_id, enrolled_on,`
`LAG(enrolled_on) OVER (PARTITION BY student_id ORDER BY enrolled_on) AS previous_enrolment`
`FROM enrolments;`
	Each enrolment next to the same student's previous enrolment date (NULL for the first).

`SELECT s.name, e.score,`
`e.score - AVG(e.score) OVER (PARTITION BY e.course_id) AS vs_course_avg`
`FROM enrolments e`
`JOIN students s ON s.id = e.student_id`
`WHERE e.score IS NOT NULL;`
	Each score compared with its course average, without collapsing rows as GROUP BY would.

Insert, update and delete safely

`INSERT INTO students (name, email, city, joined_on)` `VALUES ('Kiran', 'kiran@example.com', 'Bengaluru', CURDATE());`	Adds one student; always list the columns.
`SELECT * FROM enrolments` `WHERE status = 'active' AND enrolled_on < '2025-01-01';`	Step 1: preview exactly which rows the next UPDATE will change.

`START TRANSACTION;`
`UPDATE enrolments`
`SET status = 'dropped'`
`WHERE status = 'active' AND enrolled_on < '2025-01-01';`
`-- check the "rows affected" count, then`
`COMMIT; -- or ROLLBACK;`
	Step 2: the same WHERE in a transaction, so you can roll back if the count looks wrong.

`UPDATE enrolments e`
`JOIN courses c ON c.id = e.course_id`
`SET e.status = 'completed'`
`WHERE c.title = 'SEO Foundation' AND e.score IS NOT NULL;`
	MySQL multi-table UPDATE: marks scored SEO Foundation enrolments as completed.

`DELETE e`
`FROM enrolments e`
`JOIN students s ON s.id = e.student_id`
`WHERE s.email = 'ravi@example.com';`
	MySQL multi-table DELETE: removes every enrolment for one student.

`INSERT INTO enrolments (student_id, course_id, enrolled_on, status)`
`VALUES (3, 1, CURDATE(), 'active') AS new`
`ON DUPLICATE KEY UPDATE status = new.status, enrolled_on = new.enrolled_on;`
	Upsert: inserts the enrolment, or updates it if the student is already on that course (the unique key). Row alias syntax needs MySQL 8.0.19+.

`INSERT INTO enrolments (student_id, course_id, enrolled_on, status)`
`VALUES (3, 1, CURDATE(), 'active')`
`ON DUPLICATE KEY UPDATE status = VALUES(status);`
	The same upsert in the older VALUES() form, used by MariaDB. MySQL still accepts it but marks it deprecated.

`INSERT IGNORE INTO enrolments (student_id, course_id, enrolled_on) VALUES (1, 1, CURDATE());`	Skips the row instead of failing when it would duplicate a unique key (other errors become warnings too).

Dates

`SELECT CURDATE(), NOW();`	Today's date and the current date and time on the server.
`SELECT name FROM students WHERE joined_on >= CURDATE() - INTERVAL 30 DAY;`	Students who joined in the last 30 days.

`SELECT COUNT(*)`
`FROM enrolments`
`WHERE enrolled_on >= '2026-01-01'`
`AND enrolled_on < '2026-02-01';`
	Enrolments in January 2026. A half-open range keeps any index on the column usable.

`SELECT DATE_FORMAT(enrolled_on, '%Y-%m') AS month, COUNT(*) AS enrolments FROM enrolments GROUP BY month ORDER BY month;`	Enrolments per month, such as 2026-01, 2026-02.
`SELECT DATE_FORMAT(joined_on, '%d %b %Y') FROM students;`	Dates shown as "05 Jan 2026".
`SELECT name, DATEDIFF(CURDATE(), joined_on) AS days_since_joining FROM students;`	Days between joining and today.
`SELECT name, TIMESTAMPDIFF(MONTH, joined_on, CURDATE()) AS months FROM students;`	Whole months since joining.
`SELECT e.student_id, c.title, e.enrolled_on + INTERVAL c.duration_weeks WEEK AS expected_end FROM enrolments e JOIN courses c ON c.id = e.course_id WHERE e.status = 'active';`	Expected finish date of each active enrolment.
`SELECT LAST_DAY(CURDATE()), DAYNAME(CURDATE());`	Last day of this month and today's weekday name.

Online version, with the matching free guide: skillnewton.com/resources/cheat-sheets/sql-query-examples/
Learn this hands-on: Advanced PHP Web Development & Full-Stack Application Development – skillnewton.com/courses/advanced-php-course/