Web Programming

Database & SQL

Fenerbahce University

Course Roadmap: SQL with PHP and MySQL

100-slide path from relational data to PHP database pages

01

SQL basics

- tables, columns, rows
- SELECT, WHERE, ORDER BY

02

Data modeling

- keys and relationships
- normalization and joins

03

PHP integration

- PDO connection
- GET/POST forms + prepared queries

04

Project patterns

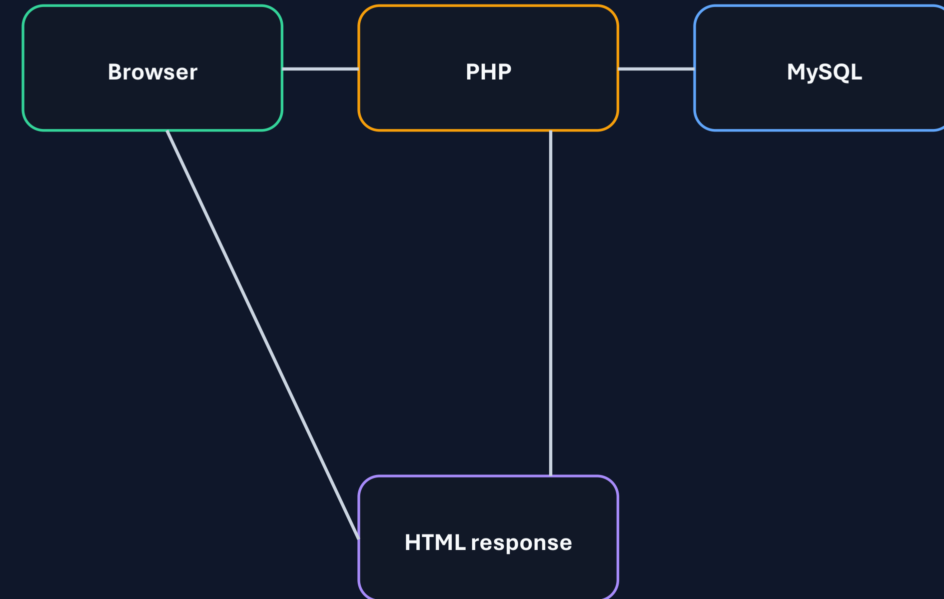
- login, reports, transactions
- admin panels and APIs

Goal: students can build a simple PHP/MySQL web application with safe database access.

How SQL Fits in a Web Application

BIG PICTURE

- HTML form collects user input.
- PHP receives GET or POST data.
- PHP sends SQL to MySQL.
- MySQL returns rows or changes data.
- PHP echoes a new HTML page.



Request → Query → Data → Rendered HTML

SQL is not a separate island: it becomes useful when PHP uses it to read and write application data.

Database Vocabulary

CONCEPT

- Database: a container for related tables.
- Table: one subject, such as students or products.
- Column: one field, such as email or price.
- Row: one record in a table.
- Query: an instruction written in SQL.

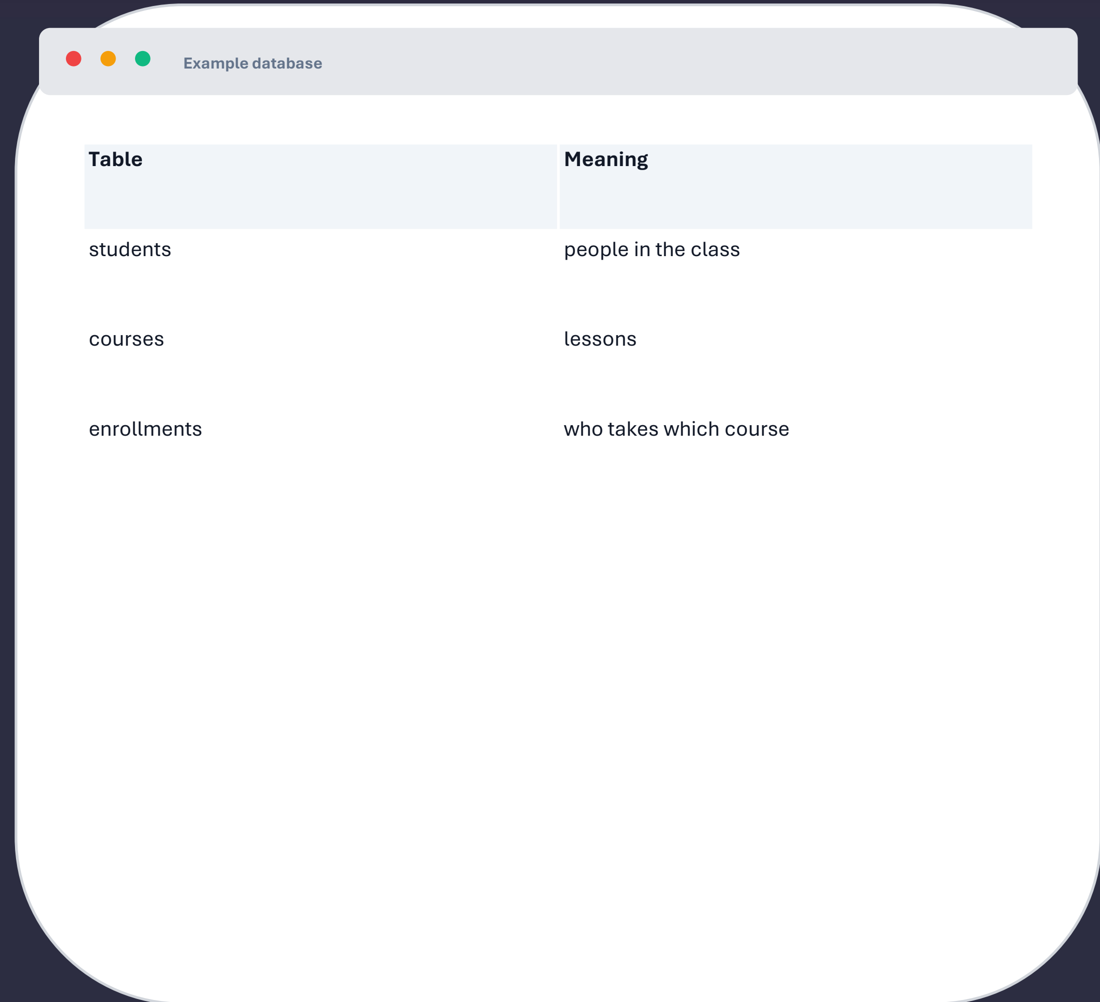


Table	Meaning
students	people in the class
courses	lessons
enrollments	who takes which course

A good database starts with clear table names and clear column names.

Sample Tables Used in Examples

CONCEPT

- students: basic student records.
- courses: course catalog.
- enrollments: connection table.
- products: small shop examples.
- orders: transaction examples.

students	courses	enrollments
id	id	student_id
first_name	title	course_id
email	credits	grade

Many examples reuse the same small data model so students can see patterns repeat in different contexts.

Working Environment

CONCEPT

- Use a local server package such as XAMPP or similar.
- Run Apache for PHP pages.
- Run MySQL for database storage.
- Use phpMyAdmin or MySQL Workbench to test SQL.
- Open PHP pages through localhost, not file://.

Local setup

Local workflow
1. Start Apache + MySQL
2. Create database
3. Write .php files
4. Test in browser

PHP must run through a web server when it needs to connect to MySQL.

SQL Syntax Rules

SQL

```sql
-- SQL keywords are usually written uppercase
SELECT first_name, email
FROM students
WHERE active = 1;

-- Semicolon ends the statement
-- Table and column names should be clear
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: students
  id | first | email          | active
* 1  | Ayse  | ayse@mail.com | 1
* 2  | Mert  | mert@mail.com | 1
  3  | Elif  | elif@mail.com | 0
```

SQL RESULT

```
first_name | email
-----------+--------------
Ayse       | ayse@mail.com
Mert       | mert@mail.com
```

SQL is readable when each clause is placed on its own line.

Create a Database

SQL

```sql
CREATE DATABASE web_course
  CHARACTER SET utf8mb4
  COLLATE utf8mb4_unicode_ci;

USE web_course;
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
Before: no database selected
Action creates: web_course
Then USE web_course selects it for queries.
```

EXPECTED DB RESULT

```
Database created: web_course
Character set: utf8mb4
Next SQL statements run inside web_course.
```

utf8mb4 is preferred because it supports Turkish characters and emojis.

Create a First Table

SQL

```
CREATE TABLE students (
  id INT AUTO_INCREMENT PRIMARY KEY,
  first_name VARCHAR(50) NOT NULL,
  last_name VARCHAR(50) NOT NULL,
  email VARCHAR(120) UNIQUE,
  phone VARCHAR(30) NULL,
  grade INT DEFAULT 0,
  active TINYINT(1) DEFAULT 1
);
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
Database: web_course
Before: no students table
Action: CREATE TABLE students
Important columns: id, email, grade, active
```

EXPECTED DB RESULT

```
AFTER
students table exists
Primary key: id
Unique field: email
Columns for examples: phone, grade, active
```

Each table should have a primary key so PHP can find exactly one row.

Insert Rows

SQL

```
INSERT INTO students
(first_name, last_name, email, phone, grade, active)
VALUES
('Ayse', 'Demir', 'ayse@mail.com', '555-111', 85, 1),
('Mert', 'Kaya', 'mert@mail.com', NULL, 68, 1),
('Elif', 'Yilmaz', 'elif@mail.com', NULL, 92, 0);
```

DB snapshot + query effect

DB BEFORE / SOURCE ROWS

```
Before INSERT
students table is empty.
AUTO_INCREMENT will generate id values.
```

AFTER / BROWSER RESULT

```
AFTER INSERT
id | first | grade | active
1  | Ayse  | 85    | 1
2  | Mert  | 68    | 1
3  | Elif  | 92    | 0
```

INSERT creates data that PHP pages can later display or update.

Select All Columns

SQL

```
SELECT *
FROM students;
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: students
  id | first | last    | grade | active | phone
* 1  | Ayse  | Demir   | 85    | 1      | 555-111
* 2  | Mert  | Kaya    | 68    | 1      | NULL
* 3  | Elif  | Yilmaz  | 92    | 0      | NULL
```

SQL RESULT

```
id | first_name | last_name | grade
---+------------+-----------+------
1  | Ayse       | Demir     | 85
2  | Mert       | Kaya      | 68
3  | Elif       | Yilmaz    | 92
```

SELECT * is useful while learning, but real applications should request only the columns they need.

Select Specific Columns

SQL

```
SELECT first_name, email
FROM students;
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: students
  id | first | last    | grade | active  | phone
* 1  | Ayse  | Demir   | 85    | 1       | 555-111
* 2  | Mert  | Kaya    | 68    | 1       | NULL
* 3  | Elif  | Yilmaz  | 92    | 0       | NULL
```

SQL RESULT

```
first_name | email
-----------+--------------
Ayse       | ayse@mail.com
Mert       | mert@mail.com
Elif       | elif@mail.com
```

Selecting fewer columns makes PHP output simpler and can improve performance.

Column Aliases

SQL

```
SELECT
  first_name AS name,
  grade AS final_score
FROM students;
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: students
  id | first | last    | grade | active | phone
* 1  | Ayse  | Demir   | 85    | 1      | 555-111
* 2  | Mert  | Kaya    | 68    | 1      | NULL
* 3  | Elif  | Yilmaz  | 92    | 0      | NULL
```

SQL RESULT

```
name | final_score
-----+------------
Ayse | 85
Mert | 68
Elif | 92
```

Aliases make column names friendlier in reports and admin screens.

WHERE with Numbers

SQL

```
SELECT first_name, grade
FROM students
WHERE grade >= 70;
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: students
  id | first | grade
* 1  | Ayse  | 85
  2  | Mert  | 68
* 3  | Elif  | 92
```

SQL RESULT

```
first_name | grade
-----------+------
Ayse       | 85
Elif       | 92
```

WHERE filters rows before PHP receives them.

WHERE with Text

SQL

```sql
SELECT first_name, email
FROM students
WHERE last_name = 'Kaya';
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: students
  id | first | last    | email
  1  | Ayse  | Demir   | ayse@mail.com
* 2  | Mert  | Kaya    | mert@mail.com
  3  | Elif  | Yilmaz  | elif@mail.com
```

SQL RESULT

```
first_name | email
-----------+--------------
Mert       | mert@mail.com
```

Text values use quotes; user-entered text must be sent through prepared statements in PHP.

AND / OR Conditions

SQL

```
SELECT first_name, grade
FROM students
WHERE grade >= 70
  AND last_name <> 'Kaya';
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: students
  id | first | grade
* 1  | Ayse  | 85
  2  | Mert  | 68
* 3  | Elif  | 92
```

SQL RESULT

```
first_name | grade
-----------+------
Ayse       | 85
Elif       | 92
```

Combine conditions to build more precise PHP search pages.

BETWEEN and IN

SQL

```
SELECT first_name, grade
FROM students
WHERE grade BETWEEN 70 AND 90
  OR id IN (2, 3);
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: students
  id | first | grade
* 1  | Ayse  | 85
  2  | Mert  | 68
* 3  | Elif  | 92
```

SQL RESULT

```
first_name | grade
-----------+------
Ayse       | 85
Mert       | 68
Elif       | 92
```

IN is common when PHP receives a selected list of IDs from checkboxes or filters.

LIKE Search

SQL

```
SELECT first_name, email
FROM students
WHERE first_name LIKE 'A%';
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: students
  id | first | email
* 1  | Ayse  | ayse@mail.com
  2  | Mert  | mert@mail.com
  3  | Elif  | elif@mail.com
```

SQL RESULT

```
first_name | email
-----------+--------------
Ayse       | ayse@mail.com
```

LIKE becomes more powerful when PHP builds the pattern from a GET search form.

Working with NULL

SQL

```
SELECT first_name, phone
FROM students
WHERE phone IS NULL;
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: students
  id | first | phone
  1  | Ayse  | 555-111
* 2  | Mert  | NULL
* 3  | Elif  | NULL
```

SQL RESULT

```
first_name | phone
-----------+------
Mert       | NULL
Elif       | NULL
```

Use IS NULL or IS NOT NULL; do not compare NULL with =.

ORDER BY

SQL

```
SELECT first_name, grade
FROM students
ORDER BY grade DESC;
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: students
  id | first | last    | grade | active | phone
* 1  | Ayse  | Demir   | 85    | 1      | 555-111
* 2  | Mert  | Kaya    | 68    | 1      | NULL
* 3  | Elif  | Yilmaz  | 92    | 0      | NULL
```

SQL RESULT

```
first_name | grade
-----------+------
Elif       | 92
Ayse       | 85
Mert       | 68
```

ORDER BY controls the order of rows before they are printed in HTML.

LIMIT and OFFSET

SQL

```
SELECT first_name, grade
FROM students
ORDER BY id
LIMIT 2 OFFSET 0;
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: students
  id | first | last    | grade | active | phone
* 1  | Ayse  | Demir   | 85    | 1      | 555-111
* 2  | Mert  | Kaya    | 68    | 1      | NULL
* 3  | Elif  | Yilmaz  | 92    | 0      | NULL
```

SQL RESULT

```
first_name | grade
-----------+------
Ayse       | 85
Mert       | 68
```

LIMIT and OFFSET are the foundation of pagination in PHP.

DISTINCT Values

SQL

```sql
SELECT DISTINCT grade
FROM students
ORDER BY grade DESC;
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: students
  id | first | last   | grade | active | phone
* 1  | Ayse  | Demir  | 85    | 1      | 555-111
* 2  | Mert  | Kaya   | 68    | 1      | NULL
* 3  | Elif  | Yilmaz | 92    | 0      | NULL
```

SQL RESULT

```
grade
-----
92
85
68
```

DISTINCT removes duplicate values and is useful for filter lists.

Calculated Columns

SQL

```sql
SELECT
  first_name,
  grade,
  grade + 5 AS bonus_grade
FROM students;
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: students
  id | first | last    | grade | active | phone
* 1  | Ayse  | Demir   | 85    | 1      | 555-111
* 2  | Mert  | Kaya    | 68    | 1      | NULL
* 3  | Elif  | Yilmaz  | 92    | 0      | NULL
```

SQL RESULT

```
first_name | grade | bonus_grade
-----------+-------+------------
Ayse       | 85    | 90
Mert       | 68    | 73
Elif       | 92    | 97
```

Calculated columns let SQL prepare values before PHP displays them.

Update a Row

SQL

```
UPDATE students
SET grade = 75
WHERE id = 2;

SELECT first_name, grade
FROM students
WHERE id = 2;
```

DB snapshot + query effect

DB BEFORE / SOURCE ROWS

```
BEFORE
students.id=2 -> Mert grade=68

SQL changes only row id=2 because WHERE id=2.
```

AFTER / BROWSER RESULT

```
AFTER UPDATE
students.id=2 -> Mert grade=75
Query result:
first_name | grade
Mert       | 75
```

Always use WHERE with UPDATE; otherwise every row may change.

Delete a Row

SQL

```sql
DELETE FROM students
WHERE id = 3;

SELECT id, first_name
FROM students;
```

DB snapshot + query effect

DB BEFORE / SOURCE ROWS

```
BEFORE
students: 1 Ayse, 2 Mert, 3 Elif

SQL removes only id=3 because WHERE id=3.
```

AFTER / BROWSER RESULT

```
AFTER DELETE
id | first_name
1  | Ayse
2  | Mert
Elif row no longer appears.
```

DELETE should usually happen after a confirmation screen in PHP.

CRUD Summary

PATTERN

- Create: INSERT from a POST form.
- Read: SELECT for listing and detail pages.
- Update: UPDATE after editing a row.
- Delete: DELETE after confirmation.
- Every admin panel is mostly CRUD.

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
Same DB table through four actions
POST form -> INSERT
List page -> SELECT
Edit form -> UPDATE
Confirm page -> DELETE
```

SQL RESULT

```
Most PHP database pages are CRUD pages.
```

When students understand CRUD, they can build many database-backed web pages.

Common MySQL Data Types

SQL

```sql
CREATE TABLE products (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  price DECIMAL(10,2) NOT NULL,
  in_stock TINYINT(1) DEFAULT 1,
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB design decision
name: text -> VARCHAR
price: money -> DECIMAL
in_stock: yes/no -> TINYINT
created_at: timestamp -> DATETIME
```

EXPECTED DB RESULT

```
Column     | Example
-----------+-------------------
name       | Keyboard
price      | 1250.00
in_stock   | 1
created_at | 2026-05-31 10:0...
```

Choose data types based on the values the PHP application must store.

Primary Key

SQL

```
CREATE TABLE courses (
  id INT AUTO_INCREMENT PRIMARY KEY,
  title VARCHAR(100) NOT NULL,
  credits INT NOT NULL
);
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
Before: no courses table
After CREATE + sample INSERTs
Each course receives one unique id.
```

EXPECTED DB RESULT

```
id | title            | credits
---+------------------+--------
1  | Web Programming  | 4
2  | Database Systems | 3
```

Primary keys are used in PHP URLs such as edit.php?id=2.

NOT NULL and DEFAULT

SQL

```sql
CREATE TABLE tasks (
  id INT AUTO_INCREMENT PRIMARY KEY,
  title VARCHAR(120) NOT NULL,
  status VARCHAR(20) NOT NULL DEFAULT 'open'
);

INSERT INTO tasks (title)
VALUES ('Check lab submissions');
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
Form sends only title.
DB fills missing status using DEFAULT open.
```

SQL RESULT

```
id | title                | status
---+--------------------+-------
1  | Check lab submi... | open
```

DEFAULT values reduce the amount of data a form must send.

UNIQUE Constraint

SQL

```
CREATE TABLE users (
  id INT AUTO_INCREMENT PRIMARY KEY,
  email VARCHAR(120) NOT NULL UNIQUE,
  password_hash VARCHAR(255) NOT NULL
);
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: users
Existing row: ayse@mail.com
New insert tries: ayse@mail.com again
UNIQUE blocks duplicate email.
```

EXPECTED DB RESULT

```
Second insert fails.
DB error: duplicate email.
PHP should show:
Email is already registered.
```

UNIQUE protects important fields such as email, username or student number.

Foreign Keys

SQL

```
CREATE TABLE enrollments (
  student_id INT NOT NULL,
  course_id INT NOT NULL,
  grade INT,
  PRIMARY KEY (student_id, course_id),
  FOREIGN KEY (student_id) REFERENCES students(id),
  FOREIGN KEY (course_id) REFERENCES courses(id)
);
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
Referenced rows must exist
students: ids 1,2
courses: id 1
enrollments can use only valid ids.
```

EXPECTED DB RESULT

```
student_id | course_id | grade
-----------+-----------+------
1          | 1         | 85
2          | 1         | 75
```

Foreign keys keep relationships valid and prevent orphan records.

Table Relationships

DATA MODEL

- One student can have many enrollments.
- One course can have many enrollments.
- The enrollments table connects them.
- This is a many-to-many relationship.
- PHP uses IDs to create and read the connection.

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
students.id -> enrollments.student_id
courses.id  -> enrollments.course_id
One student can join many courses.
```

SQL RESULT

```
One student can enroll in many courses.
```

Correct relationships make joins possible and keep the application data consistent.

INNER JOIN

SQL

```
SELECT s.first_name, c.title, e.grade
FROM enrollments e
INNER JOIN students s ON e.student_id = s.id
INNER JOIN courses c ON e.course_id = c.id;
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB rows used
students:     1 Ayse, 2 Mert
courses:      1 Web Programming, 2 Database Systems
enrollments: (1,1,85), (2,1,75)
```

SQL RESULT

```
first_name | title           | grade
-----------+-----------------+------
Ayse       | Web Programming | 85
Mert       | Web Programming | 75
```

JOIN combines rows from related tables so PHP can print meaningful names instead of IDs.

LEFT JOIN

SQL

```
SELECT c.title, e.student_id
FROM courses c
LEFT JOIN enrollments e ON c.id = e.course_id
ORDER BY c.id;
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB rows used
courses:     1 Web Programming, 2 Database Systems
enrollments: (1,1,85), (2,1,75)
No row exists for course_id=2
```

SQL RESULT

```
title            | student_id
-----------------+-----------
Web Programming  | 1
Web Programming  | 2
Database Systems | NULL
```

LEFT JOIN keeps rows from the left table even when there is no match.

Joining Three Tables

SQL

```
SELECT s.first_name, c.title, e.grade
FROM students s
JOIN enrollments e ON s.id = e.student_id
JOIN courses c ON c.id = e.course_id
WHERE c.title = 'Web Programming';
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB rows used
students:    1 Ayse, 2 Mert
courses:     1 Web Programming, 2 Database Systems
enrollments: (1,1,85), (2,1,75)
```

SQL RESULT

```
first_name | title           | grade
-----------+-----------------+------
Ayse       | Web Programming | 85
Mert       | Web Programming | 75
```

Three-table joins are common in dashboards and profile pages.

Table Aliases

SQL

```
SELECT p.name, c.name AS category
FROM products p
JOIN categories c ON p.category_id = c.id
ORDER BY p.name;
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB rows used
products:   Keyboard cat=1, Mouse cat=1, Notebook cat=2
categories: 1 Electronics, 2 Stationery
```

SQL RESULT

```
name     | category
---------+------------
Keyboard | Electronics
Notebook | Stationery
Mouse    | Electronics
```

Aliases keep SQL shorter and easier to read when multiple tables are used.

GROUP BY with COUNT

SQL

```
SELECT c.title, COUNT(e.student_id) AS total_students
FROM courses c
LEFT JOIN enrollments e ON c.id = e.course_id
GROUP BY c.id, c.title;
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB rows used
courses:     1 Web Programming, 2 Database Systems
enrollments: (1,1,85), (2,1,75)
Course 2 has no enrollment rows.
```

SQL RESULT

```
title            | total_students
-----------------+---------------
Web Programming  | 2
Database Systems | 0
```

GROUP BY creates summary rows for reports and dashboards.

SUM and AVG

SQL

```sql
SELECT
  course_id,
  AVG(grade) AS average_grade,
  MAX(grade) AS best_grade
FROM enrollments
GROUP BY course_id;
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB rows used
enrollments
course_id | student_id | grade
1         | 1          | 85
1         | 2          | 75
2         | 3          | 91
```

SQL RESULT

```
course_id | average_grade | best_grade
----------+---------------+-----------
1         | 80.00         | 85
2         | 91.00         | 91
```

Aggregates let PHP display reports without calculating everything in a loop.

HAVING

SQL

```
SELECT course_id, AVG(grade) AS average_grade
FROM enrollments
GROUP BY course_id
HAVING AVG(grade) >= 80;
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB rows used
enrollments
course_id | student_id | grade
1         | 1          | 85
1         | 2          | 75
2         | 3          | 91
```

SQL RESULT

```
course_id | average_grade
----------+--------------
1         | 80.00
2         | 91.00
```

WHERE filters rows before grouping; HAVING filters groups after grouping.

Subquery in WHERE

SQL

```
SELECT first_name, grade
FROM students
WHERE grade > (
  SELECT AVG(grade)
  FROM students
);
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: students
  id | first | grade
* 1  | Ayse  | 85
  2  | Mert  | 68
* 3  | Elif  | 92
Average grade = 81.67
```

SQL RESULT

```
first_name | grade
-----------+------
Ayse       | 85
Elif       | 92
```

Subqueries help compare a row with a calculated value.

Subquery in SELECT

SQL

```
SELECT
  c.title,
  (SELECT COUNT(*)
   FROM enrollments e
   WHERE e.course_id = c.id) AS total
FROM courses c;
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB rows used
courses:     1 Web Programming, 2 Database Systems
enrollments: (1,1,85), (2,1,75)
Course 2 has no enrollment rows.
```

SQL RESULT

```
title            | total
-----------------+------
Web Programming  | 2
Database Systems | 0
```

Subqueries in SELECT can add small summary values to each row.

UNION

SQL

```
SELECT email FROM students
UNION
SELECT email FROM teachers
ORDER BY email;
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB rows used
students.email: ayse@mail.com, mert@mail.com
teachers.email: teacher@mail.com, mert@mail.com
UNION removes duplicate emails.
```

SQL RESULT

```
email
----------------
ayse@mail.com
mert@mail.com
teacher@mail.com
```

UNION combines compatible result sets and removes duplicates.

CASE Expression

SQL

```
SELECT first_name, grade,
  CASE
    WHEN grade >= 85 THEN 'Excellent'
    WHEN grade >= 70 THEN 'Passed'
    ELSE 'Needs practice'
  END AS result
FROM students;
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: students
  first | grade
* Ayse  | 85
* Mert  | 75
* Can   | 62
CASE maps each grade to a label.
```

SQL RESULT

```
first_name | grade | result
-----------+-------+---------------
Ayse       | 85    | Excellent
Mert       | 75    | Passed
Can        | 62    | Needs practice
```

CASE prepares labels that PHP can echo directly in reports.

Date Filters

SQL

```
SELECT title, due_date
FROM assignments
WHERE due_date >= CURDATE()
ORDER BY due_date;
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: assignments
  title         | due_date
  HTML Review   | 2026-05-20
* SQL Lab       | 2026-06-05
* Final Project | 2026-06-20
CURDATE() = 2026-06-01
```

SQL RESULT

```
title         | due_date
--------------+-----------
SQL Lab       | 2026-06-05
Final Project | 2026-06-20
```

Date filters are used for upcoming events, appointments and deadlines.

String Functions

SQL

```sql
SELECT
  UPPER(first_name) AS upper_name,
  CONCAT(first_name, ' ', last_name) AS full_name
FROM students;
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: students
  first | last
* Ayse  | Demir
* Mert  | Kaya
```

SQL RESULT

```
upper_name | full_name
-----------+-----------
AYSE       | Ayse Demir
MERT       | Mert Kaya
```

SQL string functions can prepare display values before PHP prints them.

Dashboard Aggregate Query

SQL

```
SELECT
  COUNT(*) AS total_students,
  AVG(grade) AS average_grade,
  MIN(grade) AS lowest_grade,
  MAX(grade) AS highest_grade
FROM students;
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: students
  id | first | grade
* 1  | Ayse  | 85
* 2  | Mert  | 68
* 3  | Elif  | 92
```

SQL RESULT

```
total_students | average_grade | lowest_grade | highest_grade
3              | 81.67         | 68           | 92
```

A dashboard card often comes from a single aggregate SQL query.

Normalization

CONCEPT

- Do not repeat the same category name in every product row.
- Move repeated data to a separate table.
- Store the category ID in products.
- Use JOIN to display the category name again.
- This reduces update mistakes.

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
Before: repeated course data in students rows
After: students, courses, enrollments tables
Repeated text becomes related IDs.
```

SQL RESULT

```
Before               | After
---------------------+-------------------
product.categor...   | categories.id +...
repeated text        | foreign key
hard to update       | clean relationship
```

Normalization is about storing each fact in one clear place.

Many-to-Many Pivot Table

SQL

```
CREATE TABLE student_clubs (
  student_id INT NOT NULL,
  club_id INT NOT NULL,
  joined_at DATE NOT NULL,
  PRIMARY KEY (student_id, club_id),
  FOREIGN KEY (student_id) REFERENCES students(id),
  FOREIGN KEY (club_id) REFERENCES clubs(id)
);
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB rows used
students: 1 Ayse, 2 Mert
clubs: 2 Robotics, 4 Music
student_clubs: (1,2), (1,4), (2,2)
```

SQL RESULT

```
student_id | club_id | joined_at
-----------+---------+-----------
1          | 2       | 2026-03-01
1          | 4       | 2026-04-12
2          | 2       | 2026-04-20
```

A pivot table is the standard solution for many-to-many relationships.

Indexes

SQL

```sql
CREATE INDEX idx_students_email
ON students(email);

CREATE INDEX idx_products_category
ON products(category_id);
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: students
Search column: email
Without index: many rows scanned
With index: DB finds email faster.
```

SQL RESULT

```
Indexes help MySQL find rows faster.
Useful columns:
• email used in login
• foreign keys used in joins
• columns used in WHERE
```

Indexes improve reading speed, but too many indexes can slow down inserts and updates.

EXPLAIN Basics

SQL

```
EXPLAIN
SELECT *
FROM students
WHERE email = 'ayse@mail.com';
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: students
Search column: email
Without index: many rows scanned
With index: DB finds email faster.
```

SQL RESULT

```
table    | type  | key                 | rows
---------+-------+---------------------+-----
students | const | idx_students_email  | 1
```

EXPLAIN shows how MySQL plans to execute a query.

SQL Injection Problem

SQL

```sql
-- Dangerous string building
SELECT * FROM users
WHERE email = '$email'
  AND password = '$password';

-- If input changes the query,
-- attackers can bypass the logic.
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: users
Stored email: admin@mail.com
Attacker input tries to change WHERE logic
Prepared statements keep value separate.
```

SQL RESULT

```
Never place raw form values directly inside SQL.
Use prepared statements in PHP PDO.
Validate input first, then bind values.
```

Security is part of database programming, not an optional extra.

Prepared Statement Idea

SQL

```
-- Query template
SELECT * FROM users
WHERE email = ?;

-- Value is sent separately
-- email = ayse@mail.com
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: users
Stored email: admin@mail.com
Attacker input tries to change WHERE logic
Prepared statements keep value separate.
```

SQL RESULT

```
SQL text and user value travel separately.
MySQL treats the value as data, not SQL code.
This is the habit students should use in PHP.
```

Prepared statements are the bridge between SQL knowledge and safe PHP applications.

PDO Connection File

DB.PHP

```php
<?php
// db.php
$host = "localhost";
$dbname = "web_course";
$user = "root";
$pass = "";

$dsn = "mysql:host=$host;dbname=$dbname;charset=utf8mb4";
$db = new PDO($dsn, $user, $pass);
$db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

echo "Connected to database";
?>
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
Database connection target
host: localhost
dbname: web_course
user: root
PHP object: $db (PDO)
```

SQL RESULT

```
Connected to database
```

Put the connection in db.php so every page can reuse the same PDO object.

Handling Connection Errors

DB.PHP

```php
<?php
try {
  $db = new PDO($dsn, $user, $pass);
  $db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
  echo "Database is ready";
} catch (PDOException $e) {
  echo "Connection failed";
}
?>
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
Database connection target
host: localhost
dbname: web_course
user: root
PHP object: $db (PDO)
```

PHP / BROWSER OUTPUT

```
Database is ready
```

Show simple messages to users; keep detailed errors for developers.

Fetch Rows with PHP

LIST.PHP

```php
<?php
require "db.php";
$stmt = $db->query("SELECT first_name, grade FROM students");
$students = $stmt->fetchAll(PDO::FETCH_ASSOC);

foreach ($students as $student) {
  echo $student["first_name"] . " - ";
  echo $student["grade"] . "<br>";
}
?>
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: students
  id | first | grade
* 1  | Ayse  | 85
* 2  | Mert  | 75
* 3  | Elif  | 92
```

PHP / BROWSER OUTPUT

```
Ayse - 85
Mert - 75
Elif - 92
```

A SELECT result usually becomes a PHP array that can be looped with foreach.

Echo a Single Value

COUNT.PHP

```
<?php
require "db.php";
$stmt = $db->query("SELECT COUNT(*) AS total FROM students");
$row = $stmt->fetch(PDO::FETCH_ASSOC);

$total = $row["total"];
echo "Total students: " . $total;
?>
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: students
  id | first | grade
* 1  | Ayse  | 85
* 2  | Mert  | 75
* 3  | Elif  | 92
```

SQL RESULT

```
Total students: 3
```

When displaying a PHP variable, use echo so students clearly see where output happens.

PHP Generates an HTML Table

STUDENTS.PHP

```
<?php
require "db.php";
$stmt = $db->query("SELECT first_name, email FROM students");
$rows = $stmt->fetchAll(PDO::FETCH_ASSOC);

echo "<table border='1'>";
echo "<tr><th>Name</th><th>Email</th></tr>";
foreach ($rows as $row) {
  echo "<tr>";
  echo "<td>" . htmlspecialchars($row["first_name"]) . "</td>";
  echo "<td>" . htmlspecialchars($row["email"]) . "</td>";
  echo "</tr>";
}
echo "</table>";
?>
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: students
  id | first | last   | grade | active | phone
* 1  | Ayse  | Demir  | 85    | 1      | 555-111
* 2  | Mert  | Kaya   | 68    | 1      | NULL
* 3  | Elif  | Yilmaz | 92    | 0      | NULL
```

PHP / BROWSER OUTPUT

```
Name | Email
-----+--------------
Ayse | ayse@mail.com
Mert | mert@mail.com
Elif | elif@mail.com
```

PHP can echo HTML tags; the database rows become visible table rows.

GET Search Form

SEARCH.PHP

```
<form method="get">
  <input name="q" placeholder="Student name">
  <button>Search</button>
</form>

<?php
$q = $_GET["q"] ?? "";
echo "Search text: " . htmlspecialchars($q);
?>
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: students
  id | first | last   | grade | active | phone
* 1  | Ayse  | Demir  | 85    | 1      | 555-111
* 2  | Mert  | Kaya   | 68    | 1      | NULL
* 3  | Elif  | Yilmaz | 92    | 0      | NULL
```

PHP / BROWSER OUTPUT

```
Student name: [Ay]  [Search]

Search text: Ay
```

GET is suitable for search and filter forms because the query appears in the URL.

Search with LIKE and Prepared Statement

SEARCH.PHP

```
<?php
require "db.php";
$q = $_GET["q"] ?? "";

$stmt = $db->prepare(
  "SELECT first_name, email FROM students WHERE first_name LIKE ?"
);
$stmt->execute(["%" . $q . "%"]);

foreach ($stmt as $row) {
  echo htmlspecialchars($row["first_name"]) . "<br>";
}
?>
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: students
  id | first | email
* 1  | Ayse  | ayse@mail.com
  2  | Mert  | mert@mail.com
  3  | Elif  | elif@mail.com
```

PHP / BROWSER OUTPUT

```
URL: search.php?q=Ay

Ayse
```

The search text comes from GET, but the SQL value is still protected with a prepared statement.

POST Insert Form

NEW_STUDENT.PHP

```
<form method="post">
  <input name="first_name" placeholder="First name">
  <input name="email" placeholder="Email">
  <button>Save</button>
</form>

<?php
if ($_SERVER["REQUEST_METHOD"] === "POST") {
  $name = trim($_POST["first_name"] ?? "");
  $email = trim($_POST["email"] ?? "");
  echo "Received: " . htmlspecialchars($name);
}
?>
```

DB snapshot + query effect

DB BEFORE / SOURCE ROWS

```
HTML form data
first_name=Can
email=can@mail.com
PHP validates values before INSERT.
```

AFTER / BROWSER RESULT

```
First name: [Can]
Email: [can@mail.com] [Save]

Received: Can
```

POST is preferred when the form changes data in the database.

Server-Side Validation

VALIDATE.PHP

```php
<?php
$errors = [];
$name = trim($_POST["first_name"] ?? "");
$email = trim($_POST["email"] ?? "");

if ($name === "") {
  $errors[] = "Name is required.";
}
if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
  $errors[] = "Email is not valid.";
}

foreach ($errors as $error) {
  echo "<p>" . htmlspecialchars($error) . "</p>";
}
?>
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
HTML form data
first_name=Can
email=can@mail.com
PHP validates values before INSERT.
```

SQL RESULT

```
Name is required.
Email is not valid.
```

Do not trust only browser validation; PHP must validate before INSERT or UPDATE.

Insert Row with Prepared Statement

SAVE_STUDENT.PHP

```
<?php
require "db.php";
$name = trim($_POST["first_name"] ?? "");
$email = trim($_POST["email"] ?? "");

if ($name !== "" && filter_var($email, FILTER_VALIDATE_EMAIL)) {
  $stmt = $db->prepare(
    "INSERT INTO students (first_name, email) VALUES (?, ?)"
  );
  $stmt->execute([$name, $email]);
  echo "Student saved.";
}
?>
```

DB snapshot + query effect

DB BEFORE / SOURCE ROWS

```
BEFORE
students: Ayse, Mert, Elif

POST values: Can Aydin, can@mail.com
Prepared INSERT adds one new row.
```

AFTER / BROWSER RESULT

```
AFTER INSERT
id | first | email
4  | Can   | can@mail.com
Browser: Student saved.
```

Prepared INSERT protects form data and keeps the SQL structure safe.

Redirect After Saving

CREATE_CATEGORY.PHP

```
<?php
require "db.php";

if ($_SERVER["REQUEST_METHOD"] === "POST") {
  $stmt = $db->prepare("INSERT INTO categories (name) VALUES (?)");
  $stmt->execute([trim($_POST["name"])]);

  header("Location: categories.php?created=1");
  exit;
}
?>
```

DB snapshot + query effect

DB BEFORE / SOURCE ROWS

```
BEFORE
students: Ayse, Mert, Elif

POST values: Can Aydin, can@mail.com
Prepared INSERT adds one new row.
```

AFTER / BROWSER RESULT

```
AFTER INSERT
Student is saved first.
Then PHP redirects to index.php.
List page now includes the new student.
```

Redirect after POST prevents duplicate inserts when the user refreshes the browser.

Edit Page Uses GET id

EDIT_STUDENT.PHP

```php
<?php
require "db.php";
$id = (int)($_GET["id"] ?? 0);

$stmt = $db->prepare("SELECT * FROM students WHERE id = ?");
$stmt->execute([$id]);
$student = $stmt->fetch(PDO::FETCH_ASSOC);

echo "Editing: " . htmlspecialchars($student["first_name"]);
?>
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: students
id=2 is Mert Kaya
GET URL: edit.php?id=2
PHP SELECTs this row for the edit form.
```

SQL RESULT

```
URL: edit_student.php?id=2

Editing: Mert
```

GET id selects which row will be edited; POST sends the new values.

Update Row with POST

UPDATE_GRADE.PHP

```php
<?php
require "db.php";
$id = (int)($_POST["id"] ?? 0);
$grade = (int)($_POST["grade"] ?? 0);

$stmt = $db->prepare(
  "UPDATE students SET grade = ? WHERE id = ?"
);
$stmt->execute([$grade, $id]);

echo "Grade updated.";
?>
```

DB snapshot + query effect

DB BEFORE / SOURCE ROWS

```
BEFORE
students.id=2 grade=75

POST: id=2, grade=82
Prepared UPDATE changes only id=2.
```

AFTER / BROWSER RESULT

```
AFTER UPDATE
students.id=2 grade=82
Browser: Grade updated.
```

Use the row id in WHERE so only one record is changed.

Delete with Confirmation

DELETE_TASK.PHP

```
<?php
require "db.php";
$id = (int)($_POST["id"] ?? 0);

if ($_POST["confirm"] === "yes") {
  $stmt = $db->prepare("DELETE FROM tasks WHERE id = ?");
  $stmt->execute([$id]);
  echo "Task deleted.";
}
?>
<form method="post">
  <input type="hidden" name="id" value="4">
  <button name="confirm" value="yes">Delete</button>
</form>
```

DB snapshot + query effect

DB BEFORE / SOURCE ROWS

```
BEFORE
students: 1 Ayse, 2 Mert, 3 Elif
Confirm URL uses id=3
DELETE removes id=3 after confirmation.
```

AFTER / BROWSER RESULT

```
AFTER DELETE
students left: Ayse, Mert
Browser redirects to list page.
```

A confirmation step protects users from accidental DELETE operations.

Pagination with LIMIT and OFFSET

POSTS.PHP

```
<?php
require "db.php";
$page = max(1, (int)($_GET["page"] ?? 1));
$perPage = 5;
$offset = ($page - 1) * $perPage;

$stmt = $db->prepare(
  "SELECT title FROM posts ORDER BY id DESC LIMIT ? OFFSET ?"
);
$stmt->execute([$perPage, $offset]);

foreach ($stmt as $row) {
  echo htmlspecialchars($row["title"]) . "<br>";
}
?>
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: students
  id | first | last    | grade | active | phone
* 1  | Ayse  | Demir   | 85    | 1      | 555-111
* 2  | Mert  | Kaya    | 68    | 1      | NULL
* 3  | Elif  | Yilmaz  | 92    | 0      | NULL
```

SQL RESULT

```
URL: posts.php?page=2

Post 6
Post 7
Post 8
Post 9
Post 10
```

Pagination keeps large tables usable and reduces the amount of data printed in one page.

Sorting with a Whitelist

PRODUCTS.PHP

```
<?php
require "db.php";
$allowed = ["name", "price", "created_at"];
$sort = $_GET["sort"] ?? "name";

if (!in_array($sort, $allowed)) {
  $sort = "name";
}

$stmt = $db->query("SELECT name, price FROM products ORDER BY $sort");
foreach ($stmt as $row) {
  echo htmlspecialchars($row["name"]) . "<br>";
}
?>
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: students
  id | first | last   | grade | active | phone
* 1  | Ayse  | Demir  | 85    | 1      | 555-111
* 2  | Mert  | Kaya   | 68    | 1      | NULL
* 3  | Elif  | Yilmaz | 92    | 0      | NULL
```

SQL RESULT

```
URL: products.php?sort=price

Mouse
Keyboard
Monitor
```

Column names cannot be bound with ?; whitelist them before using them in ORDER BY.

Category Filter Dropdown

PRODUCTS.PHP

```
<form method="get">
  <select name="category_id">
    <option value="1">Electronics</option>
    <option value="2">Stationery</option>
  </select>
  <button>Filter</button>
</form>
<?php
require "db.php";
$cat = (int)($_GET["category_id"] ?? 0);
$stmt = $db->prepare("SELECT name FROM products WHERE category_id = ?");
$stmt->execute([$cat]);
foreach ($stmt as $row) {
  echo htmlspecialchars($row["name"]) . "<br>";
}
?>
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB rows used
products:   Keyboard cat=1, Mouse cat=1, Notebook cat=2
categories: 1 Electronics, 2 Stationery
```

SQL RESULT

```
Category: Electronics [Filter]

Keyboard
Mouse
```

GET filters are visible, shareable and safe when values are bound with prepared statements.

Count Rows in PHP Dashboard

DASHBOARD.PHP

```
<?php
require "db.php";
$stmt = $db->query("SELECT COUNT(*) AS total FROM orders");
$row = $stmt->fetch(PDO::FETCH_ASSOC);

$totalOrders = $row["total"];
echo "Orders today: " . $totalOrders;
?>
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: orders
18 rows match today
COUNT(*) returns one summary value.
```

PHP / BROWSER OUTPUT

```
Orders today: 18
```

Small dashboard cards are often simple aggregate queries echoed by PHP.

Transaction: Order + Items

CHECKOUT.PHP

```php
<?php
require "db.php";
$db->beginTransaction();

$db->prepare("INSERT INTO orders (customer_name) VALUES (?)")
   ->execute(["Ayse Demir"]);
$orderId = $db->lastInsertId();

$db->prepare(
  "INSERT INTO order_items (order_id, product_id, quantity) VALUES (?, ?, ?)"
)->execute([$orderId, 3, 2]);

$db->commit();
echo "Order saved: " . $orderId;
?>
```

DB snapshot + query effect

DB BEFORE / SOURCE ROWS

```
BEFORE
orders max id=11
order_items has no rows for id=12
Transaction will create order + item together.
```

AFTER / BROWSER RESULT

```
AFTER COMMIT
orders: id=12 Ayse Demir
order_items: order_id=12 product_id=3 qty=2
Browser: Order saved: 12
```

Transactions keep related inserts together: either all succeed or all are rolled back.

File Upload Metadata in Database

UPLOAD.PHP

```
<?php
require "db.php";
if ($_FILES["photo"]["error"] === UPLOAD_ERR_OK) {
  $name = $_FILES["photo"]["name"];
  $path = "uploads/" . basename($name);
  move_uploaded_file($_FILES["photo"]["tmp_name"], $path);

  $stmt = $db->prepare("INSERT INTO files (file_name, file_path) VALUES (?, ?)");
  $stmt->execute([$name, $path]);
  echo "File saved.";
}
?>
```

DB snapshot + query effect

DB BEFORE / SOURCE ROWS

```
Uploaded file already moved to /uploads.
DB stores only metadata: file name, size, path, user_id.
```

AFTER / BROWSER RESULT

```
AFTER INSERT
files table stores path + original name.
Actual file stays in uploads folder.
```

Store the file itself on disk; store file information in MySQL.

Login Table Design

SQL

```
CREATE TABLE users (
  id INT AUTO_INCREMENT PRIMARY KEY,
  email VARCHAR(120) NOT NULL UNIQUE,
  password_hash VARCHAR(255) NOT NULL,
  role VARCHAR(20) NOT NULL DEFAULT 'user',
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: users
  id | email            | role  | theme
* 5  | admin@mail.com   | admin | light
  6  | user@mail.com    | user  | dark
```

PHP / BROWSER OUTPUT

```
email            | password_hash | role
-----------------+---------------+------
admin@mail.com   | $2y$10$...    | admin
student@mail.com | $2y$10$...    | user
```

Never store plain text passwords in MySQL.

Register User with password_hash

REGISTER.PHP

```php
<?php
require "db.php";
$email = trim($_POST["email"] ?? "");
$password = $_POST["password"] ?? "";

$hash = password_hash($password, PASSWORD_DEFAULT);
$stmt = $db->prepare(
  "INSERT INTO users (email, password_hash) VALUES (?, ?)"
);
$stmt->execute([$email, $hash]);

echo "User registered.";
?>
```

DB snapshot + query effect

DB BEFORE / SOURCE ROWS

```
DB table: users
  id | email            | role  | theme
* 5  | admin@mail.com   | admin | light
  6  | user@mail.com    | user  | dark
```

AFTER / BROWSER RESULT

```
AFTER INSERT
users.email = new@mail.com
users.password_hash = $2y$10$...
Plain password is not stored.
```

PHP creates the hash; MySQL stores the hash string.

Login with Session

LOGIN.PHP

```
<?php
session_start();
require "db.php";

$stmt = $db->prepare("SELECT * FROM users WHERE email = ?");
$stmt->execute([$_POST["email"]]);
$user = $stmt->fetch(PDO::FETCH_ASSOC);

if ($user && password_verify($_POST["password"], $user["password_hash"])) {
  $_SESSION["user_id"] = $user["id"];
  echo "Welcome";
} else {
  echo "Invalid login";
}
?>
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: users
  id | email             | role  | theme
* 5  | admin@mail.com    | admin | light
  6  | user@mail.com     | user  | dark
```

PHP / BROWSER OUTPUT

```
Welcome
```

Database identifies the user; session remembers the user after login.

Protect an Admin Page

ADMIN.PHP

```
<?php
session_start();
if (!isset($_SESSION["user_id"])) {
  header("Location: login.php");
  exit;
}

echo "Admin panel";
?>
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: users
  id | email            | role  | theme
* 5  | admin@mail.com   | admin | light
  6  | user@mail.com    | user  | dark
```

PHP / BROWSER OUTPUT

```
Admin panel
```

A protected page checks the session before it shows database data.

Role-Based Query

ADMIN.PHP

```
<?php
session_start();
require "db.php";

$stmt = $db->prepare("SELECT role FROM users WHERE id = ?");
$stmt->execute([$_SESSION["user_id"]]);
$role = $stmt->fetchColumn();

if ($role === "admin") {
  echo "Show management buttons";
} else {
  echo "Read-only view";
}
?>
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: users
  id | email             | role  | theme
* 5  | admin@mail.com    | admin | light
  6  | user@mail.com     | user  | dark
```

PHP / BROWSER OUTPUT

```
Show management buttons
```

Roles in the database can control what PHP shows on the page.

Saving a User Preference

SETTINGS.PHP

```
<?php
require "db.php";
$userId = 5;
$theme = $_POST["theme"] ?? "light";

$stmt = $db->prepare(
  "UPDATE users SET preferred_theme = ? WHERE id = ?"
);
$stmt->execute([$theme, $userId]);

echo "Preference saved: " . htmlspecialchars($theme);
?>
```

DB snapshot + query effect

DB BEFORE / SOURCE ROWS

```
DB table: users
  id | email             | role  | theme
* 5  | admin@mail.com    | admin | light
  6  | user@mail.com     | user  | dark
```

AFTER / BROWSER RESULT

```
AFTER UPDATE
users.id=5 preferred_theme=dark
Browser: Preference saved: dark
```

Application settings can be stored in MySQL so they follow the user across browsers.

Contact Messages Table

CONTACT.PHP

```
<?php
require "db.php";
$name = trim($_POST["name"] ?? "");
$message = trim($_POST["message"] ?? "");

$stmt = $db->prepare(
  "INSERT INTO contact_messages (name, message, status) VALUES (?, ?, 'new')"
);
$stmt->execute([$name, $message]);

echo "Message received.";
?>
```

DB snapshot + query effect

DB BEFORE / SOURCE ROWS

```
BEFORE
contact_messages is empty for this message.
POST: name=Ayse, message=Need help.
```

AFTER / BROWSER RESULT

```
AFTER INSERT
contact_messages: Ayse / Need help / new
Browser: Message received.
```

A simple contact form becomes useful when messages are saved and managed in a database.

Ticket List with Status Filter

TICKETS.PHP

```
<?php
require "db.php";
$status = $_GET["status"] ?? "open";
$stmt = $db->prepare(
  "SELECT subject, status FROM tickets WHERE status = ? ORDER BY id DESC"
);
$stmt->execute([$status]);

foreach ($stmt as $ticket) {
  echo htmlspecialchars($ticket["subject"]) . " - ";
  echo htmlspecialchars($ticket["status"]) . "<br>";
}
?>
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB rows used
tickets: 1 Printer problem open
         2 Login issue open
         3 WiFi issue closed
comments for #3: issue started, restarted
```

SQL RESULT

```
URL: tickets.php?status=open

Printer problem - open
Login issue - open
```

Status columns make it easy to build filters, dashboards and admin workflows.

Ticket Detail with Comments

TICKET_DETAIL.PHP

```
<?php
require "db.php";
$ticketId = (int)($_GET["id"] ?? 0);
$stmt = $db->prepare(
  "SELECT comment_text FROM ticket_comments WHERE ticket_id = ? ORDER BY id"
);
$stmt->execute([$ticketId]);

foreach ($stmt as $comment) {
  echo "<p>" . htmlspecialchars($comment["comment_text"]) . "</p>";
}
?>
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB rows used
tickets: 1 Printer problem open
         2 Login issue open
         3 WiFi issue closed
comments for #3: issue started, restarted
```

SQL RESULT

```
Ticket #3

The issue started today.
I restarted the computer.
```

A detail page often uses one ID and then lists related rows from another table.

Appointment Booking Table

SQL

```sql
CREATE TABLE appointments (
  id INT AUTO_INCREMENT PRIMARY KEY,
  student_name VARCHAR(100) NOT NULL,
  appointment_date DATE NOT NULL,
  appointment_time TIME NOT NULL,
  note TEXT
);
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: appointments
  id | student | date         | time
* 1  | Ayse    | 2026-06-10 | 10:00
  2  | Mert    | 2026-06-10 | 10:30
Requested: 2026-06-10 10:00
```

AFTER / BROWSER RESULT

```
student_name | appointment_date | appointment_time
-------------+------------------+-----------------
Ayse         | 2026-06-10       | 10:00
Mert         | 2026-06-10       | 10:30
```

Dates and times should be stored in date/time columns, not only as text.

Prevent Double Booking

BOOK.PHP

```
<?php
require "db.php";
$date = $_POST["date"];
$time = $_POST["time"];

$stmt = $db->prepare(
  "SELECT COUNT(*) FROM appointments WHERE appointment_date = ? AND appointment_time = ?"
);
$stmt->execute([$date, $time]);

if ($stmt->fetchColumn() > 0) {
  echo "This time is already booked.";
} else {
  echo "Time is available.";
}
?>
```

DB snapshot + query effect

DB BEFORE / SOURCE ROWS

```
DB table: appointments
  id | student | date       | time
* 1  | Ayse    | 2026-06-10 | 10:00
  2  | Mert    | 2026-06-10 | 10:30
Requested: 2026-06-10 10:00
```

AFTER / BROWSER RESULT

```
COUNT result = 1
Existing row found for selected time.
Browser: This time is already booked.
```

Before INSERT, PHP can ask MySQL whether the selected time is available.

Inventory Table

SQL

```
CREATE TABLE inventory (
  id INT AUTO_INCREMENT PRIMARY KEY,
  product_name VARCHAR(100) NOT NULL,
  quantity INT NOT NULL DEFAULT 0,
  minimum_quantity INT NOT NULL DEFAULT 5
);

INSERT INTO inventory (product_name, quantity)
VALUES ('Keyboard', 12), ('Mouse', 3);
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: inventory
* product  | qty | min
* Keyboard | 12  | 5
* Mouse    | 3   | 5
```

SQL RESULT

```
product_name | quantity | minimum_quantity
-------------+----------+-----------------
Keyboard     | 12       | 5
Mouse        | 3        | 5
```

Inventory examples combine SELECT, UPDATE and report queries.

Low Stock Report

SQL

```
SELECT product_name, quantity, minimum_quantity
FROM inventory
WHERE quantity < minimum_quantity;
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: inventory
  product   | qty | min
  Keyboard | 12  | 5
* Mouse     | 3   | 5
```

SQL RESULT

```
product_name | quantity | minimum_quantity
-------------+----------+-----------------
Mouse        | 3        | 5
```

Reports are just SELECT queries with business rules in WHERE.

Orders and Order Items

SQL

```
SELECT o.id, o.customer_name, p.name, oi.quantity
FROM orders o
JOIN order_items oi ON o.id = oi.order_id
JOIN products p ON p.id = oi.product_id
WHERE o.id = 12;
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB rows used
orders: 12 Ayse Demir
order_items: (12, Mouse, 2), (12, Keyboard, 1)
products: Mouse, Keyboard
```

SQL RESULT

```
id | customer_name | name     | quantity
---+---------------+----------+---------
12 | Ayse Demir    | Mouse    | 2
12 | Ayse Demir    | Keyboard | 1
```

Order detail pages usually need joins across orders, order_items and products.

Sales Report by Day

SQL

```
SELECT
  DATE(created_at) AS sale_day,
  SUM(total_amount) AS total_sales
FROM orders
GROUP BY DATE(created_at)
ORDER BY sale_day DESC;
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: orders
  id | created_at | total_amount
* 12 | 2026-05-31 | 3000.00
* 13 | 2026-05-31 | 1250.00
* 14 | 2026-05-30 | 1800.00
```

SQL RESULT

```
sale_day   | total_sales
-----------+------------
2026-05-31 | 4250.00
2026-05-30 | 1800.00
```

GROUP BY with dates creates management reports for PHP dashboards.

Blog Posts and Comments

SQL

```sql
SELECT p.title, COUNT(c.id) AS comments
FROM posts p
LEFT JOIN comments c ON p.id = c.post_id
GROUP BY p.id, p.title
ORDER BY p.created_at DESC;
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB rows used
posts: SQL Basics, PHP Forms, HTML Review
comments: 3 for SQL, 1 for PHP, 0 for HTML
```

SQL RESULT

```
title       | comments
------------+---------
SQL Basics  | 3
PHP Forms   | 1
HTML Review | 0
```

LEFT JOIN is useful when you still want to display posts with zero comments.

Average Rating

SQL

```
SELECT product_id, AVG(rating) AS average_rating
FROM product_reviews
GROUP BY product_id
HAVING AVG(rating) >= 4;
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: product_reviews
product | ratings
* 3     | 5, 4, 5
* 7     | 4, 4, 4.6
  8     | 2, 3
```

SQL RESULT

```
product_id | average_rating
-----------+---------------
3          | 4.70
7          | 4.20
```

Aggregate queries can create recommendation or ranking lists.

Search Page with GET Parameters

PRODUCT_SEARCH.PHP

```
<form method="get">
  <input name="q" placeholder="Product">
  <input name="min" placeholder="Min price">
  <button>Search</button>
</form>
<?php
require "db.php";
$q = $_GET["q"] ?? "";
$min = (float)($_GET["min"] ?? 0);
$stmt = $db->prepare("SELECT name, price FROM products WHERE name LIKE ? AND price >= ?");
$stmt->execute(["%$q%", $min]);
foreach ($stmt as $row) {
  echo htmlspecialchars($row["name"]) . " - " . $row["price"] . "<br>";
}
?>
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: products
  id | name     | price
* 1  | Keyboard | 1250.00
  2  | Mouse    | 450.00
  3  | Keycap   | 150.00
Search: q=key and min=500
```

SQL RESULT

```
Query result
name     | price
Keyboard | 1250.00
Browser prints only products matching q + min.
```

Multiple GET inputs can become multiple safe SQL conditions.

JSON API Endpoint

API/PRODUCTS.PHP

```php
<?php
require "db.php";
header("Content-Type: application/json");

$stmt = $db->query("SELECT id, name, price FROM products");
$products = $stmt->fetchAll(PDO::FETCH_ASSOC);

echo json_encode($products);
?>
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: products
  id | name     | price   | stock
* 1  | Keyboard | 1250.00 | 8
* 2  | Mouse    | 450.00  | 25
  3  | Monitor  | 4800.00 | 2
```

PHP / BROWSER OUTPUT


```
[
  {"id":"1","name":"Keyboard","price":"1250.00"},
  {"id":"2","name":"Mouse","price":"450.00"}
]
```


A PHP page can return JSON instead of HTML when JavaScript needs data.

JavaScript Fetches PHP Data

INDEX.PHP

```
<button id="load">Load products</button>
<div id="list"></div>
<script>
document.getElementById("load").onclick = function () {
  fetch("api/products.php")
    .then(response => response.json())
    .then(products => {
      document.getElementById("list").innerText = products.length + " products loaded";
    });
};
</script>
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: products
  id | name     | price   | stock
* 1  | Keyboard | 1250.00 | 8
* 2  | Mouse    | 450.00  | 25
  3  | Monitor  | 4800.00 | 2
```

PHP / BROWSER OUTPUT

```
api/products.php returns 2 product rows as JSON.
Browser output after fetch:
2 products loaded
```

This connects the full course: HTML button, JavaScript fetch, PHP endpoint and MySQL data.

Export CSV from MySQL

EXPORT_STUDENTS.PHP

```php
<?php
require "db.php";
header("Content-Type: text/csv");
header("Content-Disposition: attachment; filename=students.csv");

$out = fopen("php://output", "w");
fputcsv($out, ["Name", "Email"]);
$stmt = $db->query("SELECT first_name, email FROM students");
foreach ($stmt as $row) {
  fputcsv($out, [$row["first_name"], $row["email"]]);
}
?>
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB table: students
  id | first | email
* 1  | Ayse  | ayse@mail.com
* 2  | Mert  | mert@mail.com
  3  | Elif  | elif@mail.com
```

PHP / BROWSER OUTPUT

```
students.csv

Name,Email
Ayse,ayse@mail.com
Mert,mert@mail.com
```

Reports can be downloaded when PHP writes database rows as CSV.

Import Data Basics

CONCEPT

- Start with a trusted CSV file.
- Read each line in PHP.
- Validate fields before insert.
- Use prepared INSERT statements.
- Report how many rows were imported.

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
CSV rows to import
1, Zeynep, zeynep@mail.com
2, bad-email
PHP validates each row before INSERT.
```

PHP / BROWSER OUTPUT

```
Expected message
Imported rows: 24
Skipped rows: 2
```

Import is a repeated validation + insert process.

Backups and Migration

CONCEPT

- A project should include the SQL file that creates tables.
- Backups protect data from mistakes.
- Migrations describe database changes over time.
- Never submit only PHP files without database structure.
- README should explain how to import the database.

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
Database files
schema.sql -> CREATE TABLE statements
sample_data.sql -> INSERT statements
README -> import steps
```

EXPECTED DB RESULT

```
Project folder
/app
/database
  schema.sql
  sample_data.sql
README.md
```

A working PHP/MySQL project must include both source code and database setup files.

Simple Error Logging

ERROR_DEMO.PHP

```
<?php
require "db.php";
try {
  $stmt = $db->query("SELECT * FROM missing_table");
} catch (PDOException $e) {
  file_put_contents("errors.log", $e->getMessage() . PHP_EOL, FILE_APPEND);
  echo "Something went wrong.";
}
?>
```

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
DB query target
missing_table does not exist.
PDO throws an exception.
PHP logs details to errors.log.
```

PHP / BROWSER OUTPUT

```
Browser shows safe message:
Something went wrong.
errors.log receives technical PDO message.
```

Users should see a simple message; developers need logs to fix the problem.

Project Schema Checklist

CONCEPT

- Every table has a primary key.
- Foreign keys connect related tables.
- Text columns have reasonable lengths.
- Money uses DECIMAL, not FLOAT.
- Created or updated timestamps are included when useful.

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
Expected project DB
At least 3 related tables
Sample data must be included
Queries should be testable in phpMyAdmin.
```

EXPECTED DB RESULT

```
Requirement | Example
------------+--------------
Primary key | id
Foreign key | user_id
Money       | DECIMAL(10,2)
Date        | created_at
```

Before coding PHP pages, students should review the database design.

Mini Project: Student Course Panel

CONCEPT

- List students from MySQL.
- Add a student with a POST form.
- Search students with a GET form.
- Edit grades using id from GET.
- Show average grade with GROUP BY.

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
Expected project DB
At least 3 related tables
Sample data must be included
Queries should be testable in phpMyAdmin.
```

SQL RESULT

```
Page             | SQL idea
-----------------+---------------
students.php     | SELECT
new_student.php  | INSERT
edit_grade.php   | UPDATE
report.php       | AVG + GROUP BY
```

This mini project practices almost every beginner SQL + PHP pattern.

Mini Project: Shop Inventory

CONCEPT

- Products and categories use a foreign key.
- Admin can add, edit and delete products.
- Users can filter by category and search by name.
- Dashboard shows low stock items.
- Order insert uses a transaction.

DB snapshot + query effect

DB DATA USED BY THIS QUERY

```
Expected project DB
At least 3 related tables
Sample data must be included
Queries should be testable in phpMyAdmin.
```

SQL RESULT

```
Feature   | Query
----------+-------------------
Catalog   | JOIN
Search    | LIKE
Low stock | WHERE quantity ...
Checkout  | TRANSACTION
```

Inventory is a strong final practice because it connects forms, joins, reports and transactions.

What students should be able to build now

01

Write SQL

- SELECT, INSERT, UPDATE, DELETE
- filters and sorting

02

Model data

- primary and foreign keys
- many-to-many tables

03

Connect PHP

- PDO and prepared statements
- GET/POST forms

04

Build apps

- CRUD pages
- reports, login, APIs

Final challenge: build a small PHP/MySQL app with at least 3 tables, 2 forms, 1 join query and 1 report.