**Intermediate SQL**

**Question 1**

**Explanation: Counts number of students per section. Then finds section with highest and lowest enrollment.**

SELECT MAX(enrollment) AS max\_enrollment, MIN(enrollment) AS min\_enrollment

FROM (

SELECT course\_id, sec\_id, semester, year, COUNT(\*) AS enrollment

FROM takes

GROUP BY course\_id, sec\_id, semester, year

) AS section\_enrollments;

A screenshot of a computer

AI-generated content may be incorrect.

**Question 2**

**Output: Returns section(s) that had the highest number of students.**

SELECT course\_id, sec\_id, semester, year, enrollment

FROM (

SELECT course\_id, sec\_id, semester, year, COUNT(\*) AS enrollment

FROM takes

GROUP BY course\_id, sec\_id, semester, year

) AS section\_enrollments

WHERE enrollment = (

SELECT MAX(enrollment)

FROM (

SELECT COUNT(\*) AS enrollment

FROM takes

GROUP BY course\_id, sec\_id, semester, year

) AS enrollments

);

**OR**

SELECT

course\_id, sec\_id, semester, year, COUNT(\*) AS enrollment

FROM takes

GROUP BY course\_id, sec\_id, semester, year

HAVING COUNT(\*) = (

SELECT

MAX(enrollment)

FROM (

SELECT

course\_id, sec\_id, semester, year, COUNT(\*) AS enrollment

FROM takes

GROUP BY course\_id, sec\_id, semester, year

) AS enrollments

);

A screenshot of a computer

AI-generated content may be incorrect.

**Question 3**

**Explanation: Joins takes with section to count students per section. LEFT JOIN ensures sections with zero students are included.**

**Using scalar subquery**

SELECT s.course\_id, s.sec\_id, s.semester, s.year,

(SELECT COUNT(\*)

FROM takes t

WHERE t.course\_id = s.course\_id AND t.sec\_id = s.sec\_id

AND t.semester = s.semester AND t.year = s.year) AS enrollment

FROM section s;

**OR**

SELECT

s.course\_id, s.sec\_id, s.semester, s.year,

IFNULL((

SELECT COUNT(\*)

FROM takes t

WHERE t.course\_id = s.course\_id

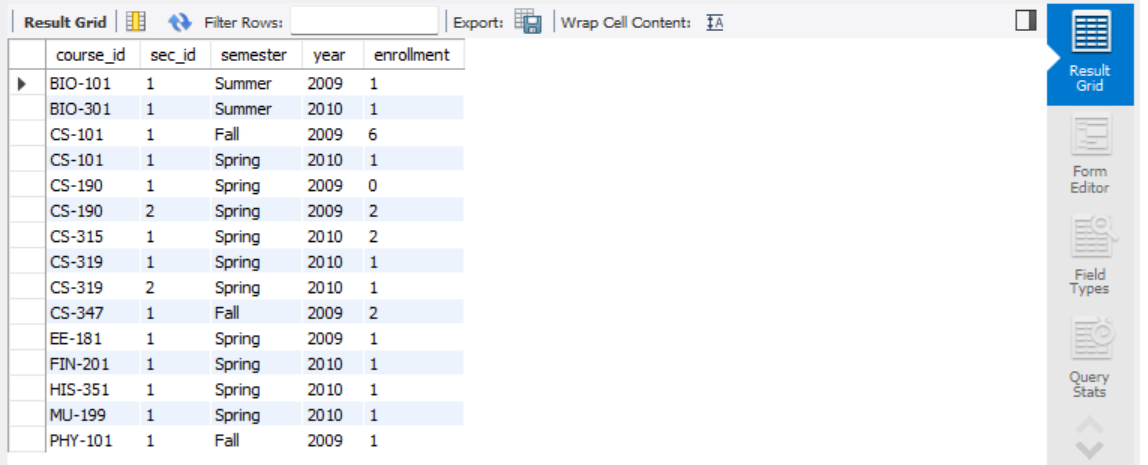
AND t.sec\_id = s.sec\_id

AND t.semester = s.semester

AND t.year = s.year

), 0) AS enrollment

FROM section s;



**Using left outer join with aggregation**

SELECT s.course\_id, s.sec\_id, s.semester, s.year,

COUNT(t.ID) AS enrollment

FROM section s

LEFT OUTER JOIN takes t

ON s.course\_id = t.course\_id AND s.sec\_id = t.sec\_id

AND s.semester = t.semester AND s.year = t.year

GROUP BY s.course\_id, s.sec\_id, s.semester, s.year;

**OR**

SELECT

s.course\_id, s.sec\_id, s.semester, s.year,

IFNULL(COUNT(t.ID), 0) AS enrollment

FROM section s

LEFT OUTER JOIN takes t

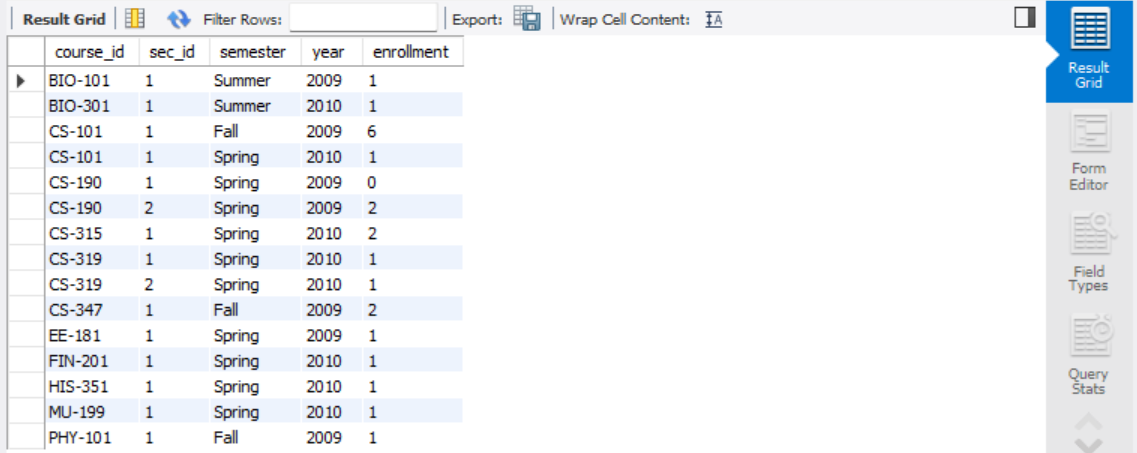
ON s.course\_id = t.course\_id

AND s.sec\_id = t.sec\_id

AND s.semester = t.semester

AND s.year = t.year

GROUP BY s.course\_id, s.sec\_id, s.semester, s.year;



**Question 4**

**Explanation: Filters course\_id that begins with 'CS-1' (e.g., CS-101, CS-102).**

SELECT \* FROM course

WHERE course\_id LIKE 'CS-1%';

A screenshot of a computer

AI-generated content may be incorrect.

**Question 5 (might need data)**

**Explanation: For each instructor, checks if there’s any 'CS-1%' course they haven’t taught. If not, include them.**

**Using NOT EXISTS… EXCEPT**

SELECT i.ID, i.name

FROM instructor i

WHERE NOT EXISTS (

SELECT c.course\_id

FROM course c

WHERE c.course\_id LIKE 'CS-1%'

EXCEPT

SELECT t.course\_id

FROM teaches t

WHERE t.ID = i.ID

);

**OR**

SELECT i.ID, i.name

FROM instructor i

WHERE NOT EXISTS (

SELECT course\_id

FROM course

WHERE course\_id LIKE 'CS-1%'

EXCEPT

SELECT t.course\_id

FROM teaches t

WHERE t.ID = i.ID

);

**Using matching counts**

SELECT i.ID, i.name

FROM instructor i, (

SELECT COUNT(DISTINCT course\_id) AS total\_cs1

FROM course

WHERE course\_id LIKE 'CS-1%'

) AS needed

WHERE (

SELECT COUNT(DISTINCT t.course\_id)

FROM teaches t

WHERE t.ID = i.ID AND t.course\_id LIKE 'CS-1%'

) = needed.total\_cs1;

**OR**

SELECT i.ID, i.name

FROM instructor i

JOIN teaches t ON i.ID = t.ID

WHERE t.course\_id LIKE 'CS-1%'

GROUP BY i.ID, i.name

HAVING COUNT(DISTINCT t.course\_id) = (

SELECT COUNT(\*)

FROM course

WHERE course\_id LIKE 'CS-1%'

);

**Question 6**

**Explanation: Adds instructors into student table. Prevents duplication using NOT IN.**

INSERT INTO student (ID, name, dept\_name, tot\_cred)

SELECT ID, name, dept\_name, 0

FROM instructor;

**OR**

INSERT INTO student (ID, name, dept\_name, tot\_cred)

SELECT ID, name, dept\_name, 0

FROM instructor

WHERE ID NOT IN (

SELECT ID FROM student

);

**Question 7**

**Explanation: Removes such students. Safe updates disabled to allow deletion with subquery.**SET SQL\_SAFE\_UPDATES = 0;

DELETE FROM student

WHERE tot\_cred = 0 AND ID IN (SELECT ID FROM instructor);

SET SQL\_SAFE\_UPDATES = 1;

**Question 8**

**Explanation: Totals credits of passed courses and updates each student's tot\_cred.**

UPDATE student s

SET tot\_cred = (

SELECT SUM(c.credits)

FROM takes t

JOIN course c ON t.course\_id = c.course\_id

WHERE t.ID = s.ID

AND t.grade IS NOT NULL

AND t.grade != 'F'

);

**OR If you prefer to set those to 0 instead of NULL, wrap it with IFNULL(..., 0):**

UPDATE student s

SET tot\_cred = IFNULL((

SELECT SUM(c.credits)

FROM takes t

JOIN course c ON t.course\_id = c.course\_id

WHERE t.ID = s.ID

AND t.grade IS NOT NULL

AND t.grade != 'F'

), 0);

**Question 9**

**Explanation: Each section taught adds 10,000 to salary.**

UPDATE instructor i

SET salary = (

SELECT COUNT(\*) \* 10000

FROM teaches t

WHERE t.ID = i.ID

);

**Advanced SQL**

**Question 1**

**Purpose: Creates a view of students who failed a course and never retook it and passed.**

**📤 Output: Rows from takes where the student got 'F' and no subsequent pass exists for the same course.**

CREATE VIEW UnoverriddenFails AS

SELECT \* FROM takes t1

WHERE grade = 'F'

AND NOT EXISTS (

SELECT 1

FROM takes t2

WHERE t2.ID = t1.ID

AND t2.course\_id = t1.course\_id

AND t2.grade IS NOT NULL

AND t2.grade != 'F'

);

**Question 2 (might need data)**

**Purpose: Lists students who have failed at least 2 different courses and never passed them.**

**📤 Output: Entries from the UnoverriddenFails view filtered for those students.**

SELECT \*

FROM UnoverriddenFails

WHERE ID IN (

SELECT ID

FROM UnoverriddenFails

GROUP BY ID

HAVING COUNT(\*) >= 2

);

**Question 3**

**Output: Student ID, name, and CPI (Cumulative Performance Index). Students with no grades return NULL CPI (or use IFNULL(..., 0) if desired).**

CREATE TABLE grade\_points (

grade VARCHAR(2) PRIMARY KEY,

point INT

);

INSERT INTO grade\_points (grade, point) VALUES

('A', 10), ('A-', 9), ('B+', 8), ('B', 8),

('B-', 7), ('C+', 6), ('C', 6),

('C-', 5), ('D', 4), ('F', 0);

SELECT s.ID, s.name,

ROUND(SUM(gp.point \* c.credits) / SUM(c.credits), 2) AS CPI

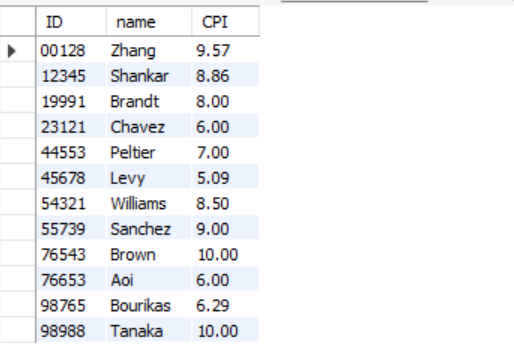
FROM student s

JOIN takes t ON s.ID = t.ID

JOIN course c ON t.course\_id = c.course\_id

JOIN grade\_points gp ON t.grade = gp.grade

GROUP BY s.ID, s.name;



**Include All Students (Even if No Grades)- Use left outer join**

SELECT s.ID, s.name,

ROUND(SUM(gp.point \* c.credits) / SUM(c.credits), 2) AS CPI

FROM student s

LEFT JOIN takes t ON s.ID = t.ID

LEFT JOIN course c ON t.course\_id = c.course\_id

LEFT JOIN grade\_points gp ON t.grade = gp.grade

GROUP BY s.ID, s.name;

**Question 4 (might need data)**

**Purpose: Identifies course pairs scheduled in the same room, same time, same semester/year.**

**📤 Output: Conflicting course pairs (course1, course2) that overlap in schedule.**

WITH room\_assignments AS (

SELECT course\_id, sec\_id, semester, year, building, room\_number, time\_slot\_id

FROM section

)

SELECT

RA1.building, RA1.room\_number, RA1.time\_slot\_id, RA1.semester, RA1.year,

RA1.course\_id AS course1, RA2.course\_id AS course2

FROM room\_assignments RA1, room\_assignments RA2

WHERE RA1.building = RA2.building

AND RA1.room\_number = RA2.room\_number

AND RA1.time\_slot\_id = RA2.time\_slot\_id

AND RA1.semester = RA2.semester

AND RA1.year = RA2.year

AND (RA1.course\_id, RA1.sec\_id) < (RA2.course\_id, RA2.sec\_id);

**Question 5**

**Purpose: Creates a simplified view showing basic instructor info.**

**📤 Output: View showing instructor ID, name, and department.**

CREATE VIEW faculty AS

SELECT ID, name, dept\_name

FROM instructor;

**Question 6**

**Purpose: Restricts view to only Computer Science instructors. WITH CHECK OPTION ensures inserted/updated rows match the filter.**

**📤 Output: View of CS instructors only.**

CREATE VIEW CSinstructors AS

SELECT \*

FROM instructor

WHERE dept\_name = 'Comp. Sci.';

**OR (Enforce the Filter on Inserts)**

CREATE VIEW CSinstructors AS

SELECT \* FROM instructor

WHERE dept\_name = 'Comp. Sci.'

WITH CHECK OPTION;

**Question 7**

**Purpose: Demonstrates allowed and disallowed insertions into views.**

**📤 Output: First two insert; third fails due to WITH CHECK OPTION.**

INSERT INTO faculty (ID, name, dept\_name)

VALUES ('99999', 'Ada Lovelace', 'Physics'); - **succeeds**

INSERT INTO CSinstructors (ID, name, dept\_name, salary)

VALUES ('88888', 'Alan Turing', 'Comp. Sci.', 100000); - **succeds**

INSERT INTO CSinstructors (ID, name, dept\_name, salary)

VALUES ('77777', 'Marie Curie', 'Physics', 95000); - **fails(with check option) bcs falls outside the dept\_name requirement**

**Question 8**

**Purpose: Creates a new user and gives read-only access to the student table.**

**📤 Output: User created and can query student data.**

**Create user -** CREATE USER 'jane\_doe'@'localhost' IDENTIFIED BY 'securepassword';

**Grant them permission -** GRANT SELECT ON university1.student TO 'jane\_doe'@'localhost';

**Question 9**

**Purpose: Covers inter-user view sharing and access control.**

**📤 Output: Allows cross-user reading of views if proper GRANT is applied.**

1. **Grant SELECT to All Users on Your View**

GRANT SELECT ON university1.faculty TO '%'@'%';

**See if friend granted you**

SHOW GRANTS FOR CURRENT\_USER;

**OR**

SHOW GRANTS FOR 'root'@'localhost';

1. **Select From a Friend’s View**

SELECT \* FROM john\_doe.faculty;

SELECT \* FROM university1.faculty;

**Incase you don’t have permission (friend grants you) - GRANT SELECT ON university1.faculty TO 'your\_username'@'localhost';**

GRANT SELECT ON university1.faculty TO 'root'@'localhost';