**LAB-3**

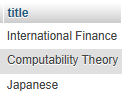
3.1 Write the following queries in SQL, using the university schema. (We suggest you actually run these queries on a database, using the sample data that we provide on the web site of the book, db-book.com. Instructions for setting up a database, and loading sample data, are provided on the above web site.)

a. Find the titles of courses in the Comp. Sci. department that have 3 credits.

SELECT title

FROM course

WHERE dept\_name = 'Comp. Sci.' AND credits = 3;



b. Find the IDs of all students who were taught by an instructor named Einstein; make sure there are no duplicates in the result.

SELECT DISTINCT takes.ID

FROM takes

JOIN teaches ON takes.course\_id = teaches.course\_id

AND takes.sec\_id = teaches.sec\_id

AND takes.semester = teaches.semester

AND takes.year = teaches.year

JOIN instructor ON teaches.ID = instructor.ID

WHERE instructor.name = 'Einstein';

NO output:

c. Find the highest salary of any instructor.

SELECT MAX(salary) AS highest\_salary

FROM instructor;



d. Find all instructors earning the highest salary (there may be more than one with the same salary).

SELECT name, salary

FROM instructor

WHERE salary = (SELECT MAX(salary) FROM instructor);



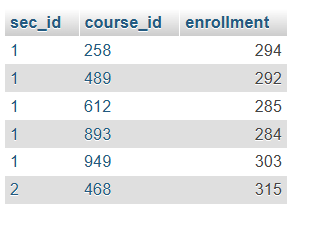
e. Find the enrollment of each section that was offered in Fall 2017.

SELECT sec\_id, course\_id, COUNT(ID) AS enrollment

FROM takes

WHERE semester = 'Fall' AND year = 2017

GROUP BY sec\_id, course\_id;



f. Find the maximum enrollment, across all sections, in Fall 2017.

SELECT MAX(enrollment) AS max\_enrollment

FROM (

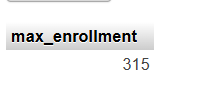
SELECT sec\_id, course\_id, COUNT(ID) AS enrollment

FROM takes

WHERE semester = 'Fall' AND year = 2017

GROUP BY sec\_id, course\_id

) AS section\_enrollment;



g. Find the sections that had the maximum enrollment in Fall 2017.

SELECT sec\_id, course\_id, COUNT(ID) AS enrollment

FROM takes

WHERE semester = 'Fall' AND year = 2007

GROUP BY sec\_id, course\_id

HAVING COUNT(ID) = (

SELECT MAX(enrollment)

FROM (

SELECT sec\_id, course\_id, COUNT(ID) AS enrollment

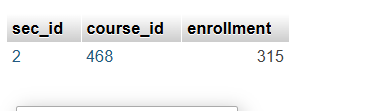
FROM takes

WHERE semester = 'Fall' AND year = 2007

GROUP BY sec\_id, course\_id

) AS section\_enrollment

);



3.2 Suppose you are given a relation grade points(grade, points) that provides a con version from letter grades in the takes relation to numeric scores; for example, an “A” grade could be specified to correspond to 4 points, an “A−” to 3.7points, a “B+” to 3.3 points, a “B” to 3 points, and so on. The grade points earned by a student for a course offering (section) is defined as the number of credits for the course multiplied by the numeric points for the grade that the student received. Given the preceding relation, and our university schema, write each of the following queries in SQL. You may assume for simplicity that no takes tuple has the null value for grade.

a. Find the total grade points earned by the student with ID '12345', across all courses taken by the student.

SELECT SUM(course.credits \* grade\_points.points) AS total\_grade\_points

FROM takes

JOIN course ON takes.course\_id = course.course\_id

JOIN grade\_points ON takes.grade = grade\_points.grade

WHERE takes.ID = '12345';

b. Find the grade point average (GPA) for the above student, that is, the total grade points divided by the total credits for the associated courses.

SELECT SUM(course.credits \* grade\_points.points) / SUM(course.credits) AS GPA

FROM takes

JOIN course ON takes.course\_id = course.course\_id

JOIN grade\_points ON takes.grade = grade\_points.grade

WHERE takes.ID = '12345';

c. Find the ID and the grade-point average of each student.

SELECT takes.ID,

SUM(course.credits \* grade\_points.points) / SUM(course.credits) AS GPA

FROM takes

JOIN course ON takes.course\_id = course.course\_id

JOIN grade\_points ON takes.grade = grade\_points.grade

GROUP BY takes.ID;

d. Now reconsider your answers to the earlier parts of this exercise under the assumption that some grades might be null. Explain whether your solutions still work and, if not, provide versions that handle nulls properly.

If some grades are NULL, the above queries would fail because multiplication with NULL results in NULL. We can handle this using COALESCE(), which replaces NULL values with 0 for grade points.

SELECT takes.ID,

SUM(course.credits \* COALESCE(grade\_points.points, 0)) / NULLIF(SUM(course.credits), 0) AS GPA

FROM takes

LEFT JOIN course ON takes.course\_id = course.course\_id

LEFT JOIN grade\_points ON takes.grade = grade\_points.grade

GROUP BY takes.ID;

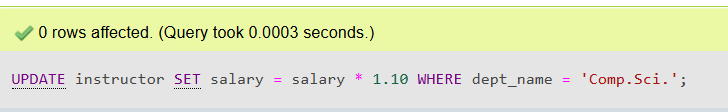
3.3 Write the following inserts, deletes, or updates in SQL, using the university schema.

a. Increase the salary of each instructor in the Comp. Sci. department by 10%.

UPDATE instructor

SET salary = salary \* 1.10

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



b. Delete all courses that have never been offered (i.e., do not occur in the section relation).

DELETE FROM course

WHERE course\_id NOT IN (SELECT DISTINCT course\_id FROM section);

c. Insert every student whose tot cred attribute is greater than 100 as an instructor in the same department, with a salary of $10,000.

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

SELECT ID, name, dept\_name, 10000

FROM student

WHERE tot\_cred > 100;

3.4 Consider the insurance database of Figure 3.17, where the primary keys are underlined. Construct the following SQL queries for this relational database.

a. Find the total number of people who owned cars that were involved in accidents in 2017.

SELECT COUNT(DISTINCT owns.driver\_id) AS total\_owners

FROM owns

JOIN participated ON owns.license\_plate = participated.license\_plate

JOIN accident ON participated.report\_number = accident.report\_number

WHERE accident.year = 2017;

b. Delete all year-2010 cars belonging to the person whose ID is '12345'.

DELETE FROM car

WHERE license\_plate IN (

SELECT license\_plate FROM owns WHERE driver\_id = '12345'

) AND year = 2010;

3.5 Suppose that we have a relation marks(ID, score) and we wish to assign grades to students based on the score as follows: grade F if score < 40, grade C if 40 ≤score < 60, grade B if 60 ≤ score < 80, and grade A if 80 ≤ score. Write SQL queries to do the following:

a. Display the grade for each student, based on the marks relation.

SELECT ID, score,

CASE

WHEN score < 40 THEN 'F'

WHEN score >= 40 AND score < 60 THEN 'C'

WHEN score >= 60 AND score < 80 THEN 'B'

ELSE 'A'

END AS grade

FROM marks;

b. Find the number of students with each grade.

SELECT

CASE

WHEN score < 40 THEN 'F'

WHEN score >= 40 AND score < 60 THEN 'C'

WHEN score >= 60 AND score < 80 THEN 'B'

ELSE 'A'

END AS grade,

COUNT(\*) AS num\_students

FROM marks

GROUP BY grade;

3.6 The SQL like operator is case sensitive (in most systems), but the lower() function on strings can be used to perform case-insensitive matching. To show how, write a query that finds departments whose names contain the string “sci” as a substring, regardless of the case.

SELECT dept\_name

FROM department

WHERE LOWER(dept\_name) LIKE '%sci%';

3.7 Consider the SQL query

select p.a1

from p, r1, r2

where p.a1 = r1.a1 or p.a1 = r2.a1

Under what conditions does the preceding query select values of p. a1 that are either in r1 or in r2? Examine carefully the cases where either r1 or r2 may be empty.

To correctly return values of p.a1 that exist in **either** r1 or r2, we should use **LEFT JOIN** instead of an implicit Cartesian product:

SELECT DISTINCT p.a1

FROM p

LEFT JOIN r1 ON p.a1 = r1.a1

LEFT JOIN r2 ON p.a1 = r2.a1

WHERE r1.a1 IS NOT NULL OR r2.a1 IS NOT NULL;

3.8 Consider the bank database of Figure 3.18, where the primary keys are under lined. Construct the following SQL queries for this relational database.

(branch(branch name, branch city, assets)

customer (ID, customer name, customer street, customer city)

loan (loan number, branch name, amount)

borrower (ID, loan number)

account (account number, branch name, balance )

depositor (ID, account number) )

Figure 3.18 Banking database.

a. Find the ID of each customer of the bank who has an account but not a loan.

SELECT c.ID

FROM customer c

WHERE c.ID IN (SELECT d.ID

FROM depositor d)

AND c.ID NOT IN (SELECT b.ID

FROM borrower b);

b. Find the ID of each customer who lives on the same street and in the same city as customer '12345'.

SELECT c1.ID

FROM customer c1, customer c2

WHERE c1.customer\_street = c2.customer\_street

AND c1.customer\_city = c2.customer\_city

AND c2.ID = '12345'

AND c1.ID != '12345';

c. Find the name of each branch that has at least one customer who has an account in the bank and who lives in “Harrison”.

SELECT DISTINCT b.branch\_name

FROM branch b

WHERE EXISTS (SELECT 1

FROM customer c, depositor d

WHERE c.ID = d.ID

AND d.branch\_name = b.branch\_name

AND c.customer\_city = 'Harrison');

3.9 Consider the relational database of Figure 3.19, where the primary keys are underlined. Give an expression in SQL for each of the following queries.

(employee (ID, person name, street, city)

works (ID, company name, salary)

company (company name, city)

manages (ID, manager id))

Figure 3.19 Employee database.

a. Find the ID, name, and city of residence of each employee who works for “First Bank Corporation”.

SELECT e.ID, e.person\_name, e.city

FROM employee e

JOIN works w ON e.ID = w.ID

WHERE w.company\_name = 'First Bank Corporation';

b. Find the ID, name, and city of residence of each employee who works for “First Bank Corporation” and earns more than $10000.

SELECT e.ID, e.person\_name, e.city

FROM employee e

JOIN works w ON e.ID = w.ID

WHERE w.company\_name = 'First Bank Corporation' AND w.salary > 10000;

c. Find the ID of each employee who does not work for “First Bank Corporation”.

SELECT e.ID

FROM employee e

WHERE e.ID NOT IN (SELECT w.ID

FROM works w

WHERE w.company\_name = 'First Bank Corporation');

d. Find the ID of each employee who earns more than every employee of “Small Bank Corporation”.

SELECT e.ID

FROM employee e

WHERE e.salary > ALL (SELECT w.salary

FROM works w

WHERE w.company\_name = 'Small Bank Corporation'

AND w.ID = e.ID);

e. Assume that companies may be located in several cities. Find the name of each company that is located in every city in which “Small Bank Cor poration” is located.

SELECT c.company\_name

FROM company c

WHERE NOT EXISTS (SELECT 1

FROM company c2

WHERE c2.company\_name = 'Small Bank Corporation'

AND NOT EXISTS (SELECT 1

FROM company c3

WHERE c3.company\_name = c.company\_name

AND c3.city = c2.city));

f. Find the name of the company that has the most employees (or compa nies, in the case where there is a tie for the most).

SELECT w.company\_name

FROM works w

GROUP BY w.company\_name

HAVING COUNT(w.ID) = (SELECT MAX(employee\_count)

FROM (SELECT COUNT(ID) AS employee\_count

FROM works

GROUP BY company\_name) AS subquery);

g. Find the name of each company whose employees earn a higher salary, on average, than the average salary at “First Bank Corporation”.

SELECT w.company\_name

FROM works w

GROUP BY w.company\_name

HAVING AVG(w.salary) > (SELECT AVG(salary)

FROM works

WHERE company\_name = 'First Bank Corporation');

3.10 Consider the relational database of Figure 3.19. Give an expression in SQL for each of the following:

a. Modify the database so that the employee whose ID is '12345' now lives in “Newtown”.

UPDATE employee

SET city = 'Newtown'

WHERE ID = '12345';

b. Give each manager of “First Bank Corporation” a 10 percent raise unless the salary becomes greater than $100000; in such cases, give only a 3 percent raise.

UPDATE works

SET salary = CASE

WHEN salary \* 1.10 > 100000 THEN salary \* 1.03

ELSE salary \* 1.10

END

WHERE ID IN (SELECT DISTINCT manager\_id

FROM manages

WHERE manager\_id IN (SELECT ID

FROM employee

WHERE company\_name = 'First Bank Corporation'));

3.11 Write the following queries in SQL, using the university schema.

a. Find the ID and name of each student who has taken at least one Comp. Sci. course; make sure there are no duplicate names in the result.

SELECT DISTINCT s.ID, s.name

FROM student s

JOIN enrollment e ON s.ID = e.student\_ID

JOIN course c ON e.course\_ID = c.ID

WHERE c.department = 'Comp. Sci.';

b. Find the ID and name of each student who has not taken any course offered before 2017.

SELECT s.ID, s.name

FROM student s

WHERE NOT EXISTS (SELECT 1

FROM enrollment e

JOIN course c ON e.course\_ID = c.ID

WHERE e.student\_ID = s.ID

AND c.year < 2007);

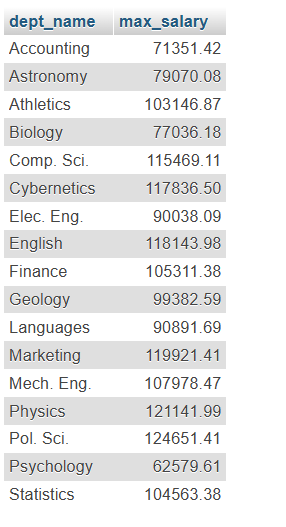
c. For each department, find the maximum salary of instructors in that department. You may assume that every department has at least one instructor.

SELECT d.dept\_name, MAX(i.salary) AS max\_salary

FROM department d

JOIN instructor i ON d.dept\_name = i.dept\_name

GROUP BY d.dept\_name;



d. Find the lowest, across all departments, of the per-department maximum salary computed by the preceding query.

SELECT MIN(max\_salary) AS lowest\_max\_salary

FROM (

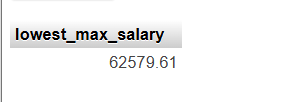
SELECT MAX(i.salary) AS max\_salary

FROM department d

JOIN instructor i ON d.dept\_name = i.dept\_name

GROUP BY d.dept\_name

) AS department\_max\_salaries;



3.12 Write the SQL statements using the university schema to perform the following operations:

a. Create a new course “CS-001”, titled “Weekly Seminar”, with 0 credits.

INSERT INTO course (course\_ID, title, credits)

VALUES ('CS-001', 'Weekly Seminar', 0);

b. Create a section of this course in Fall 2017, with sec id of 1, and with the location of this section not yet specified.

INSERT INTO section (course\_ID, sec\_id, year, term, location)

VALUES ('CS-001', 1, 2017, 'Fall', NULL);

c. Enroll every student in the Comp. Sci. department in the above section.

INSERT INTO enrollment (student\_ID, course\_ID, sec\_id)

SELECT s.ID, 'CS-001', 1

FROM student s

JOIN department d ON s.department = d.department\_name

WHERE d.department\_name = 'Comp. Sci.';

d. Delete enrollments in the above section where the student’s ID is 12345.

DELETE FROM enrollment

WHERE student\_ID = '12345' AND course\_ID = 'CS-001' AND sec\_id = 1;

e. Delete the course CS-001. What will happen if you run this delete statement without first deleting offerings (sections) of this course?

DELETE FROM course

WHERE course\_ID = 'CS-001';

f. Delete all takes tuples corresponding to any section of any course with the word “advanced” as a part of the title; ignore case when matching the word with the title.

DELETE FROM takes

WHERE section\_ID IN (

SELECT sec\_id

FROM section s

JOIN course c ON s.course\_ID = c.course\_ID

WHERE LOWER(c.title) LIKE '%advanced%'

);

3.13 Write SQL DDL corresponding to the schema in Figure 3.17.

(person (driver id, name, address)

car (license plate, model, year)

accident (report number, year, location)

owns (driver id, license plate)

participated (report number, license plate, driver id, damage amount) )

Figure 3.17 Insurance database

Make any reason able assumptions about data types, and be sure to declare primary and foreign keys.

Here is the SQL DDL corresponding to the schema in Figure 3.17 for the Insurance database:

-- Create the `person` table

CREATE TABLE person (

driver\_id INT PRIMARY KEY, -- Assuming driver\_id is an integer

name VARCHAR(100) NOT NULL, -- Name as a string, maximum 100 characters

address VARCHAR(255) -- Address as a string, maximum 255 characters

);

-- Create the `car` table

CREATE TABLE car (

license\_plate VARCHAR(20) PRIMARY KEY, -- License plate as a string (up to 20 characters)

model VARCHAR(100) NOT NULL, -- Car model as a string

year INT CHECK (year > 1900 AND year <= EXTRACT(YEAR FROM CURRENT\_DATE)) -- Year, reasonable range

);

-- Create the `accident` table

CREATE TABLE accident (

report\_number INT PRIMARY KEY, -- Report number as an integer

year INT CHECK (year > 1900 AND year <= EXTRACT(YEAR FROM CURRENT\_DATE)), -- Year of the accident

location VARCHAR(255) NOT NULL -- Location as a string

);

-- Create the `owns` table to link drivers and cars

CREATE TABLE owns (

driver\_id INT, -- Driver ID from the `person` table

license\_plate VARCHAR(20), -- License plate from the `car` table

PRIMARY KEY (driver\_id, license\_plate), -- Composite primary key

FOREIGN KEY (driver\_id) REFERENCES person(driver\_id) ON DELETE CASCADE, -- Foreign key to `person`

FOREIGN KEY (license\_plate) REFERENCES car(license\_plate) ON DELETE CASCADE -- Foreign key to `car`

);

-- Create the `participated` table to link accidents and cars with drivers, and store damage amount

CREATE TABLE participated (

report\_number INT, -- Report number from the `accident` table

license\_plate VARCHAR(20), -- License plate from the `car` table

driver\_id INT, -- Driver ID from the `person` table

damage\_amount DECIMAL(10, 2), -- Damage amount, assuming a monetary value

PRIMARY KEY (report\_number, license\_plate, driver\_id), -- Composite primary key

FOREIGN KEY (report\_number) REFERENCES accident(report\_number) ON DELETE CASCADE, -- Foreign key to `accident`

FOREIGN KEY (license\_plate) REFERENCES car(license\_plate) ON DELETE CASCADE, -- Foreign key to `car`

FOREIGN KEY (driver\_id) REFERENCES person(driver\_id) ON DELETE CASCADE -- Foreign key to `person`

);

3.14 Consider the insurance database of Figure 3.17, where the primary keys are underlined. Construct the following SQL queries for this relational database.

a. Find the number of accidents involving a car belonging to a person named “John Smith”.

SELECT COUNT(DISTINCT p.driver\_id) AS accident\_count

FROM person p

JOIN owns o ON p.driver\_id = o.driver\_id

JOIN participated pa ON o.license\_plate = pa.license\_plate

WHERE p.name = 'John Smith';

b. Update the damage amount for the car with license plate “AABB2000” in the accident with report number “AR2197” to $3000.

UPDATE participated

SET damage\_amount = 3000

WHERE license\_plate = 'AABB2000' AND report\_number = 'AR2197';

3.15 Consider the bank database of Figure 3.18, where the primary keys are under lined. Construct the following SQL queries for this relational database.

a. Find each customer who has an account at every branch located in “Brooklyn”.

SELECT c.ID, c.customer\_name

FROM customer c

WHERE NOT EXISTS (

SELECT b.branch\_name

FROM branch b

WHERE b.branch\_city = 'Brooklyn'

AND NOT EXISTS (

SELECT 1

FROM account a

WHERE a.branch\_name = b.branch\_name

AND a.customer\_id = c.ID

)

);

b. Find the total sum of all loan amounts in the bank.

SELECT SUM(l.amount) AS total\_loan\_amount

FROM loan l;

c. Find the names of all branches that have assets greater than those of at least one branch located in “Brooklyn”.

SELECT DISTINCT b.branch\_name

FROM branch b

WHERE b.assets > (

SELECT MAX(bk.assets)

FROM branch bk

WHERE bk.branch\_city = 'Brooklyn'

);

3.16 Consider the employee database of Figure 3.19, where the primary keys are underlined. Give an expression in SQL for each of the following queries.

a. Find ID and name of each employee who lives in the same city as the location of the company for which the employee works.

SELECT e.ID, e.person\_name

FROM employee e

JOIN works w ON e.ID = w.ID

JOIN company c ON w.company\_name = c.company\_name

WHERE e.city = c.city;

b. Find ID and name of each employee who lives in the same city and on the same street as does her or his manager.

SELECT e.ID, e.person\_name

FROM employee e

JOIN works w ON e.ID = w.ID

JOIN manages m ON e.ID = m.manager\_id

JOIN employee mngr ON m.manager\_id = mngr.ID

WHERE e.city = mngr.city AND e.street = mngr.street;

c. Find ID and name of each employee who earns more than the average salary of all employees of her or his company.

SELECT e.ID, e.person\_name

FROM employee e

JOIN works w ON e.ID = w.ID

WHERE w.salary > (

SELECT AVG(w2.salary)

FROM works w2

WHERE w2.company\_name = w.company\_name

);

d. Find the company that has the smallest payroll.

SELECT w.company\_name

FROM works w

GROUP BY w.company\_name

ORDER BY SUM(w.salary) ASC

LIMIT 1;

3.17 Consider the employee database of Figure 3.19. Give an expression in SQL for each of the following queries.

a. Give all employees of “First Bank Corporation” a 10 percent raise.

UPDATE works

SET salary = salary \* 1.10

WHERE company\_name = 'First Bank Corporation';

b. Give all managers of “First Bank Corporation” a 10 percent raise.

UPDATE works

SET salary = salary \* 1.10

WHERE company\_name = 'First Bank Corporation'

AND ID IN (SELECT DISTINCT manager\_id FROM manages WHERE manager\_id IS NOT NULL);

c. Delete all tuples in the works relation for employees of “Small Bank Corporation”.

DELETE FROM works

WHERE company\_name = 'Small Bank Corporation';

3.18 Give an SQL schema definition for the employee database of Figure 3.19. Choose an appropriate domain for each attribute and an appropriate primary key for each relation schema. Include any foreign-key constraints that might be appropriate.

-- Create the `employee` table

CREATE TABLE employee (

ID INT PRIMARY KEY, -- Employee ID, unique identifier

person\_name VARCHAR(100) NOT NULL, -- Name of the employee (up to 100 characters)

street VARCHAR(255), -- Street address (up to 255 characters)

city VARCHAR(100) -- City (up to 100 characters)

);

-- Create the `company` table

CREATE TABLE company (

company\_name VARCHAR(100) PRIMARY KEY, -- Company name, unique identifier

city VARCHAR(100) -- City where the company is located

);

-- Create the `works` table to link employees with companies and their salaries

CREATE TABLE works (

ID INT, -- Employee ID (foreign key referencing employee table)

company\_name VARCHAR(100), -- Company name (foreign key referencing company table)

salary DECIMAL(10, 2) NOT NULL, -- Salary (decimal with 2 decimal places)

PRIMARY KEY (ID, company\_name), -- Composite primary key (employee, company)

FOREIGN KEY (ID) REFERENCES employee(ID) ON DELETE CASCADE, -- Foreign key to employee

FOREIGN KEY (company\_name) REFERENCES company(company\_name) ON DELETE CASCADE -- Foreign key to company

);

-- Create the `manages` table to link employees with their managers

CREATE TABLE manages (

ID INT, -- Employee ID (foreign key referencing employee table)

manager\_id INT, -- Manager's employee ID (foreign key referencing employee table)

PRIMARY KEY (ID), -- Primary key is employee ID (assuming only one manager per employee)

FOREIGN KEY (ID) REFERENCES employee(ID) ON DELETE CASCADE, -- Foreign key to employee

FOREIGN KEY (manager\_id) REFERENCES employee(ID) ON DELETE CASCADE -- Foreign key to manager (also employee)

);

3.19 List two reasons why null values might be introduced into the database.

Null values can be introduced into a database for two main reasons:

1. **Unknown or Missing Data**: When information is unavailable, such as a new employee not yet providing a phone number.
2. **Not Applicable Data**: When a field does not apply to a particular record, such as a spouse name for an unmarried customer.

3.20 Show that, in SQL, <> all is identical to not in.

In SQL, <> ALL and NOT IN are logically equivalent:

* **<> ALL** ensures a value is **not equal** to any value in a list or subquery.
* **Example:**
* SELECT \*
* FROM employees
* WHERE salary <> ALL (SELECT salary FROM employees WHERE department = 'Sale
* **NOT IN** checks if a value is **not present** in a list or subquery.
* **Example:**
* SELECT \*
* FROM employees
* WHERE salary NOT IN (SELECT salary FROM employees WHERE department = 'Sales'

3.21 Consider the library database of Figure 3.20. Write the following queries in SQL.

(member(memb no, name)

book(isbn, title, authors, publisher)

borrowed(memb\_no, isbn, date))

Figure 3.20 Library database.

a. Find the member number and name of each member who has borrowed at least one book published by “McGraw-Hill”.

SELECT DISTINCT m.memb\_no, m.name

FROM member m

JOIN borrowed b ON m.memb\_no = b.memb\_no

JOIN book bo ON b.isbn = bo.isbn

WHERE bo.publisher = 'McGraw-Hill';

b. Find the member number and name of each member who has borrowed every book published by “McGraw-Hill”.

SELECT m.memb\_no, m.name

FROM member m

WHERE NOT EXISTS (

SELECT bo.isbn

FROM book bo

WHERE bo.publisher = 'McGraw-Hill'

AND NOT EXISTS (

SELECT b.isbn

FROM borrowed b

WHERE b.memb\_no = m.memb\_no

AND b.isbn = bo.isbn

)

);

c. For each publisher, find the member number and name of each member who has borrowed more than five books of that publisher.

SELECT bo.publisher, m.memb\_no, m.name

FROM member m

JOIN borrowed b ON m.memb\_no = b.memb\_no

JOIN book bo ON b.isbn = bo.isbn

GROUP BY bo.publisher, m.memb\_no, m.name

HAVING COUNT(b.isbn) > 5;

d. Find the average number of books borrowed per member. Take into account that if a member does not borrow any books, then that member does not appear in the borrowed relation at all, but that member still counts in the average.

SELECT AVG(borrowed\_books) AS avg\_books\_borrowed

FROM (

SELECT m.memb\_no, COUNT(b.isbn) AS borrowed\_books

FROM member m

LEFT JOIN borrowed b ON m.memb\_no = b.memb\_no

GROUP BY m.memb\_no

) AS member\_borrow\_counts;

3.22 Rewrite the where clause

where unique (select title from course)

without using the unique construct.

WHERE NOT EXISTS (

SELECT title

FROM course

GROUP BY title

HAVING COUNT(\*) > 1

)

3.23 Consider the query:

with dept total (dept name, value) as

(select dept name, sum(salary)

from instructor

group by dept name),

dept total avg(value) as

(select avg(value)

from dept total)

select dept name

from dept total, dept total avg

where dept total.value >= dept total avg.value;

Rewrite this query without using the with construct.

SELECT dept\_name

FROM (

SELECT dept\_name, SUM(salary) AS value

FROM instructor

GROUP BY dept\_name

) AS dept\_total,

(

SELECT AVG(value) AS value

FROM (

SELECT dept\_name, SUM(salary) AS value

FROM instructor

GROUP BY dept\_name

) AS dept\_total

) AS dept\_total\_avg

WHERE dept\_total.value >= dept\_total\_avg.value;

3.24 Using the university schema, write an SQL query to find the name and ID of those Accounting students advised by an instructor in the Physics department.

SELECT s.ID, s.name

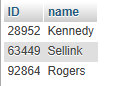
FROM student s

JOIN advisor a ON s.ID = a.s\_id

JOIN instructor i ON a.i\_id = i.ID

WHERE s.dept\_name = 'Accounting'

AND i.dept\_name = 'Physics';



3.25 Using the university schema, write an SQL query to find the names of those departments whose budget is higher than that of Philosophy. List them in alphabetic order.

SELECT dept\_name

FROM department

WHERE budget > (SELECT budget FROM department WHERE dept\_name = 'Philosophy')

ORDER BY dept\_name;

3.26 Using the university schema, use SQL to do the following: for each student who has retaken a course at least twice (i.e., the student has taken the course at least three times), show the course ID and the student’s ID. Please display your results in order of course ID and do not display duplicate rows.

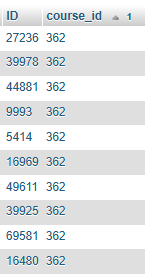
SELECT DISTINCT t.ID, t.course\_id

FROM takes t

GROUP BY t.ID, t.course\_id

HAVING COUNT(\*) >= 3

ORDER BY t.course\_id;



3.27 Using the university schema, write an SQL query to find the IDs of those students who have retaken at least three distinct courses at least once (i. e, the student has taken the course at least two times).

SELECT t.ID

FROM takes t

GROUP BY t.ID, t.course\_id

HAVING COUNT(\*) >= 2

GROUP BY t.ID

HAVING COUNT(DISTINCT t.course\_id) >= 3;



3.28 Using the university schema, write an SQL query to find the names and IDs of those instructors who teach every course taught in his or her department (i.e., every course that appears in the course relation with the instructor’s department name). Order result by name.

SELECT i.ID, i.name

FROM instructor i

WHERE NOT EXISTS (

SELECT c.course\_id

FROM course c

WHERE c.dept\_name = i.dept\_name

AND NOT EXISTS (

SELECT 1

FROM teaches t

WHERE t.course\_id = c.course\_id

AND t.ID = i.ID

)

)

ORDER BY i.name;



3.29 Using the university schema, write an SQL query to find the name and ID of each History student whose name begins with the letter ‘D’ and who has not taken at least five Music courses.

SELECT s.ID, s.name

FROM student s

WHERE s.dept\_name = 'History'

AND s.name LIKE 'D%'

AND (

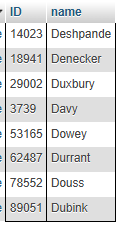
SELECT COUNT(\*)

FROM takes t

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

WHERE t.ID = s.ID AND c.dept\_name = 'Music'

) < 5;



3.30 Consider the following SQL query on the university schema:

select avg(salary)-(sum(salary)/count(\*))

from instructor

We might expect that the result of this query is zero since the average of a set of numbers is defined to be the sum of the numbers divided by the number of numbers. In deed this is true for the example instructor relation in Figure 2.1. However, there are other possible instances of that relation for which the result would not be zero. Give one such instance, and explain why the result would not be zero.

The query **AVG(salary) - (SUM(salary) / COUNT(\*))** should ideally return **zero** because both expressions mathematically compute the average salary.

However, due to **floating-point precision errors**, some database instances may store AVG(salary) and SUM(salary)/COUNT(\*) with slight differences, leading to a **small non-zero result** (e.g., 0.0000000001).

This happens because **computers store decimal numbers in binary**, which can introduce rounding errors in separate calculations.

3.31 Using the university schema, write an SQL query to find the ID and name of each instructor who has never given an A grade in any course she or he has taught. (Instructors who have never taught a course trivially satisfy this condition.)

SELECT i.ID, i.name

FROM instructor i

WHERE NOT EXISTS (

SELECT 1

FROM teaches t

JOIN takes tk ON t.course\_id = tk.course\_id

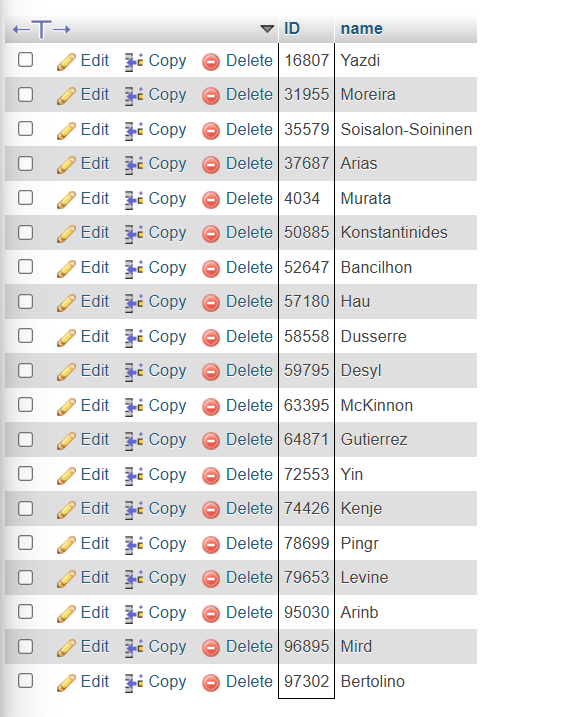
AND t.sec\_id = tk.sec\_id

AND t.semester = tk.semester

AND t.year = tk.year

WHERE t.ID = i.ID AND tk.grade = 'A'

);



3.32 Rewrite the preceding query, but also ensure that you include only instructors who have given at least one other non-null grade in some course.

SELECT i.ID, i.name

FROM instructor i

WHERE NOT EXISTS (

SELECT 1

FROM teaches t

JOIN takes tk ON t.course\_id = tk.course\_id

AND t.sec\_id = tk.sec\_id

AND t.semester = tk.semester

AND t.year = tk.year

WHERE t.ID = i.ID AND tk.grade = 'A'

)

AND EXISTS (

SELECT 1

FROM teaches t

JOIN takes tk ON t.course\_id = tk.course\_id

AND t.sec\_id = tk.sec\_id

AND t.semester = tk.semester

AND t.year = tk.year

WHERE t.ID = i.ID AND tk.grade IS NOT NULL

);

3.33 Using the university schema, write an SQL query to find the ID and title of each course in Comp. Sci. that has had at least one section with after noon hours (i.e., ends at or after 12:00). (You should eliminate duplicates if any.)

SELECT DISTINCT c.course\_id, c.title

FROM course c

JOIN section s ON c.course\_id = s.course\_id

JOIN time\_slot t ON s.time\_slot\_id = t.time\_slot\_id

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

AND t.end\_time >= '12:00';

3.34 Using the university schema, write an SQL query to find the number of students in each section. The result columns should appear in the order “course\_id, sec\_id, year, semester, num”. You do not need to output sections with 0 students.

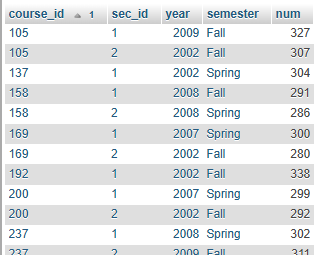
SELECT t.course\_id, t.sec\_id, t.year, t.semester, COUNT(t.ID) AS num

FROM takes t

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

HAVING COUNT(t.ID) > 0

ORDER BY t.course\_id, t.sec\_id, t.year, t.semester;



3.35 Using the university schema, write an SQL query to find section(s) with maximum enrollment. The result columns should appear in the order “courseid, secid, year, semester, num”. (It may be convenient to use the with construct.)

WITH SectionEnrollment AS (

SELECT t.course\_id, t.sec\_id, t.year, t.semester, COUNT(t.ID) AS num

FROM takes t

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

)

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

FROM SectionEnrollment s

WHERE s.num = (SELECT MAX(num) FROM SectionEnrollment);

