# PATUAKHALI SCIENCE AND TECHNOLOGY UNIVERSITY

**COURSE CODE CCE-223**

## SUBMITTED TO:


**Prof. Dr. Md Samsuzzaman**

**Department of Computer and Communication Engineering**

**Faculty of Computer Science and Engineering**


## SUBMITTED BY:


**Sanjida Islam Nuha**

ID: **2102063**

Registration No: **10190**

**Faculty of Computer Science and Engineering**


Date of submission: **8 Feb 2025**

# Practice Exercises

**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, dbbook.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;*

| ***title*** |
|---|
| ***International Finance*** |
| ***Computability Theory*** |
| ***Japanese*** |

**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 USING (course_id, sec_id, semester, year)*

*JOIN instructor ON teaches.ID = instructor.ID*

*WHERE instructor.name = 'Einstein';*

**c. *Find the highest salary of any instructor.***

*SELECT MAX(salary) AS highest_salary*

*FROM instructor;*

***124651.41***

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

*SELECT name*

*FROM instructor*

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

***Wieland***

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

```sql
SELECT course_id, sec_id, COUNT(ID) AS enrollment
FROM takes
WHERE semester = 'Fall' AND year = 2017
GROUP BY course_id, sec_id;
```

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

```sql
SELECT MAX(enrollment) AS max_enrollment
FROM (
SELECT COUNT(ID) AS enrollment
FROM takes
WHERE semester = 'Fall' AND year = 2017
GROUP BY course_id, sec_id
) AS enrollments;
```

| max_enrollment |
|---|
| NULL |

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

```sql
SELECT course_id, sec_id
FROM takes
WHERE semester = 'Fall' AND year = 2017
GROUP BY course_id, sec_id
HAVING COUNT(ID) = (
SELECT MAX(enrollment)
FROM (
SELECT COUNT(ID) AS enrollment
FROM takes
```

*WHERE semester = 'Fall' AND year = 2017*

*GROUP BY course_id, sec_id*

*) AS max_enroll*

*);*

**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 3.7 points, 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.**

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.**

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

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.**

**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_people
FROM owns
JOIN participated USING (license_plate)
JOIN accident USING (report_number)
WHERE accident.year = 2017;
```

*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

**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 < 60 THEN 'C'
WHEN 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 < 60 THEN 'C'
WHEN score < 80 THEN 'B'
ELSE 'A'
END AS grade,
COUNT(*) AS total_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.**

**Ans:**

The given SQL query:

```
SELECT p.a1
FROM p, r1, r2
WHERE p.a1 = r1.a1 OR p.a1 = r2.a1;
```

performs a **Cartesian product** of `p`, `r1`, and `r2` before applying the `WHERE` condition. This causes problems when **either `r1` or `r2` is empty**, leading to an empty result.

### Key Issues:

1. If **both `r1` and `r2` have data** → Works correctly.
2. If **either `r1` or `r2` is empty** → The query returns an empty result, even if `p.a1` exists in the other table.
3. If **both `r1` and `r2` are empty** → The query also returns an empty result.

Instead of a Cartesian product, use `LEFT JOIN`:

```
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;
```

This ensures `p.a1` is selected even if **one of the tables is empty**.

Let me know if you need more details!

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

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

*SELECT DISTINCT d.ID*

*FROM depositor d*

*LEFT JOIN borrower b ON d.ID = b.ID*

*WHERE b.ID IS NULL;*

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

*SELECT c2.ID*
*FROM customer c1*
*JOIN customer c2 ON c1.customer_street = c2.customer_street*
*AND c1.customer_city = c2.customer_city*
*WHERE c1.ID = '12345' AND c2.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 a.branch_name*
*FROM account a*
*JOIN depositor d ON a.account_number = d.account_number*
*JOIN customer c ON d.ID = c.ID*
*WHERE c.customer_city = 'Harrison';*

---

*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.

**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.**

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

*SELECT e.ID, e.name, e.city*

*FROM employee e*

*JOIN works w ON e.ID = w.ID*

```sql
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.**

```sql
SELECT e.ID, e.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”.**

```sql
SELECT DISTINCT ID
FROM works
WHERE company_name <> 'First Bank Corporation';
```

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

```sql
SELECT DISTINCT w1.ID
FROM works w1
WHERE w1.salary > ALL (
SELECT w2.salary
FROM works w2
WHERE w2.company_name = 'Small Bank Corporation'
);
```

**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 Corporation” is located.**

```sql
SELECT DISTINCT c1.company_name
FROM company c1
WHERE NOT EXISTS (
SELECT c2.city
FROM company c2
WHERE c2.company_name = 'Small Bank Corporation'
AND NOT EXISTS (
SELECT *
FROM company c3
WHERE c3.company_name = c1.company_name AND c3.city = c2.city
)
);
```

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

```
SELECT company_name
FROM works
GROUP BY company_name
HAVING COUNT(ID) = (
SELECT MAX(emp_count)
FROM (SELECT company_name, COUNT(ID) AS emp_count FROM works GROUP BY company_name)
AS company_counts
);
```

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

```
SELECT w1.company_name
FROM works w1
GROUP BY w1.company_name
HAVING AVG(w1.salary) > (
SELECT AVG(w2.salary)
FROM works w2
WHERE w2.company_name = 'First Bank Corporation'
);
```

---

*employee* (<u>*ID*</u>, *person_name*, *street*, *city*)
*works* (<u>*ID*</u>, *company_name*, *salary*)
*company* (<u>*company_name*</u>, *city*)
*manages* (<u>*ID*</u>, *manager_id*)

---

**Figure 3.19** Employee database.

**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 * 1.10
ELSE salary * 1.03
END
WHERE ID IN (SELECT ID FROM manages)
AND company_name = 'First Bank Corporation';
```

## Exercises

**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 student.ID, student.name
FROM student
JOIN takes ON student.ID = takes.ID
JOIN course ON takes.course_id = course.course_id
WHERE course.dept_name = 'Comp. Sci.';
```

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

```
SELECT student.ID, student.name
FROM student
WHERE student.ID NOT IN (
SELECT DISTINCT takes.ID
```

FROM takes

JOIN section ON takes.course_id = section.course_id AND takes.sec_id = section.sec_id

WHERE section.year < 2017

);

**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 dept_name, MAX(salary) AS max_salary

FROM instructor

GROUP BY dept_name;

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

*SELECT MIN(max_salary)*

*FROM (*

*SELECT dept_name, MAX(salary) AS max_salary*

*FROM instructor*

*GROUP BY dept_name*

*) AS dept_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, dept_name, credits)

VALUES ('CS-001', 'Weekly Seminar', 'Comp. Sci.', 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, semester, year, building, room_number)*

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

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

INSERT INTO takes (ID, course_id, sec_id, semester, year)

SELECT ID, 'CS-001', 1, 'Fall', 2017

FROM student

WHERE dept_name = 'Comp. Sci.';

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

DELETE FROM takes

WHERE ID = '12345' AND course_id = 'CS-001' AND sec_id = 1 AND semester = 'Fall' AND year = 2017;

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 course_id IN (*

*SELECT course_id*

*FROM course*

*WHERE LOWER(title) LIKE '%advanced%'*

*);*

**3.13 Write SQL DDL corresponding to the schema in Figure 3.17. Make any reasonable assumptions about data types, and be sure to declare primary and foreign keys.** *CREATE TABLE person ( driver_id INT PRIMARY KEY, name VARCHAR(100) NOT NULL, address VARCHAR(255) NOT NULL*

*);*

*CREATE TABLE car (*

*license_plate VARCHAR(20) PRIMARY*

*KEY, model VARCHAR(50) NOT NULL,*

*year INT CHECK (year >= 1886)*

*);*

*CREATE TABLE accident (*

*report_number INT PRIMARY KEY,*

*year INT CHECK (year >= 1900),*

*location VARCHAR(255) NOT NULL*

*);*

*CREATE TABLE owns (*

*driver_id INT, license_plate*

*VARCHAR(20),*

*PRIMARY KEY (driver_id, license_plate),*

*FOREIGN KEY (driver_id) REFERENCES person(driver_id) ON DELETE CASCADE,*

*FOREIGN KEY (license_plate) REFERENCES car(license_plate) ON DELETE CASCADE*

*);*

*CREATE TABLE participated (*

*report_number INT,*

*license_plate VARCHAR(20),*

*driver_id INT,*

*damage_amount DECIMAL(10,2) CHECK (damage_amount >= 0), negative*

*PRIMARY KEY (report_number, license_plate, driver_id),*

*FOREIGN KEY (report_number) REFERENCES accident(report_number) ON DELETE CASCADE,*

*FOREIGN KEY (license_plate) REFERENCES car(license_plate) ON DELETE CASCADE,*

*FOREIGN KEY (driver_id) REFERENCES person(driver_id) ON DELETE CASCADE*

*);*

**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 participated.report_number)*

*FROM participated*

*JOIN owns ON participated.license_plate = owns.license_plate*

*JOIN person ON owns.driver_id = person.driver_id*

*WHERE person.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 report_number = 'AR2197' AND license_plate = 'AABB2000';*

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

**a. Find each customer who has an account at every branch located in "Brooklyn".**

```
SELECT d.ID
FROM depositor d
JOIN account a ON d.account_number = a.account_number
JOIN branch b ON a.branch_name = b.branch_name
WHERE b.branch_city = 'Brooklyn'
GROUP BY d.ID
HAVING COUNT(DISTINCT a.branch_name) = (SELECT COUNT(branch_name) FROM branch WHERE
branch_city = 'Brooklyn');
```

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

*SELECT SUM(amount) AS total_loan_amount FROM loan;*

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

```
SELECT DISTINCT b1.branch_name
FROM branch b1
WHERE b1.assets > (
SELECT MIN(b2.assets)
FROM branch b2
WHERE b2.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 e1.ID, e1.person_name
FROM employee e1
JOIN manages m ON e1.ID = m.ID
JOIN employee e2 ON m.manager_id = e2.ID
WHERE e1.city = e2.city AND e1.street = e2.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 w1 ON e.ID = w1.ID
WHERE w1.salary > (
SELECT AVG(w2.salary)
FROM works w2
WHERE w2.company_name = w1.company_name
);
```

**d. Find the company that has the smallest payroll.**

```
SELECT company_name
FROM works
GROUP BY company_name
ORDER BY SUM(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 ID IN (
SELECT ID FROM manages
) AND company_name = 'First Bank Corporation';
```

**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 TABLE employee (
ID INT PRIMARY KEY,
person_name
VARCHAR(100), street
VARCHAR(255), city
VARCHAR(100)
);

CREATE TABLE company (
company_name VARCHAR(100) PRIMARY KEY,
city VARCHAR(100)
);
```

*CREATE TABLE works (*

*ID INT,*

*company_name VARCHAR(100),*

*salary DECIMAL(10,2),*

*PRIMARY KEY (ID, company_name),*

*FOREIGN KEY (ID) REFERENCES employee(ID),*

*FOREIGN KEY (company_name) REFERENCES company(company_name)*

*);*

*CREATE TABLE manages (*

*ID INT PRIMARY KEY,*

*manager_id INT,*

*FOREIGN KEY (ID) REFERENCES employee(ID),*

*FOREIGN KEY (manager_id) REFERENCES employee(ID)*

*);*

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

**Ans:**

1. **Missing Information:** Data might not be available at the time of entry. For example, a customer's phone number might be unknown.
2. **Not Applicable:** Some attributes may not be relevant for certain rows. For example, an employee without a manager will have a NULL `manager_id`.

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

*SELECT ID FROM employee*

*WHERE salary <> ALL (SELECT salary FROM works WHERE company_name = 'Small Bank Corporation');*

This means **ID is selected if its salary is different from every salary in "Small Bank Corporation"**, which is identical to:

*SELECT ID FROM employee*

*WHERE salary NOT IN (SELECT salary FROM works WHERE company_name = 'Small Bank Corporation');*

*member*(*memb_no*, *name*)
*book*(*isbn*, *title*, *authors*, *publisher*)
*borrowed*(*memb_no*, *isbn*, *date*)

**Figure 3.20** Library database.

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

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

```
SELECT DISTINCT member.memb_no, member.name
FROM member
JOIN borrowed ON member.memb_no = borrowed.memb_no
JOIN book ON borrowed.isbn = book.isbn
WHERE book.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 b.isbn
FROM book b
WHERE b.publisher = 'McGraw-Hill'
EXCEPT
SELECT br.isbn
FROM borrowed br
WHERE br.memb_no = m.memb_no
);
```

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

*SELECT book.publisher, borrowed.memb_no, member.name*

*FROM borrowed*

*JOIN book ON borrowed.isbn = book.isbn*

*JOIN member ON borrowed.memb_no = member.memb_no*

*GROUP BY book.publisher, borrowed.memb_no, member.name*

*HAVING COUNT(borrowed.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 COUNT(borrowed.isbn) * 1.0 / COUNT(DISTINCT member.memb_no) AS avg_books_per_member*

*FROM member*

*LEFT JOIN borrowed ON member.memb_no = borrowed.memb_no;*

**3.22 Rewrite the where clause where unique (select title from course) without using the unique construct.**

*WHERE (SELECT COUNT(DISTINCT title) FROM course) = 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*

*JOIN (*

*SELECT AVG(value) AS avg_value*

*FROM (*

*SELECT dept_name, SUM(salary) AS value*

*FROM instructor*

*GROUP BY dept_name*

*) AS dept_total_avg*

*) AS overall_avg*

*ON dept_total.value >= overall_avg.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 student.ID, student.name*

*FROM student*

*JOIN advisor ON student.ID = advisor.s_id*

*JOIN instructor ON advisor.i_id = instructor.ID*

*WHERE student.dept_name = 'Accounting' AND instructor.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 course_id, ID*

```
FROM takes
GROUP BY course_id, ID
HAVING COUNT(*) >= 3
ORDER BY course_id, 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 ID
FROM takes
GROUP BY ID, course_id
HAVING COUNT(*) >= 2
GROUP BY ID
HAVING COUNT(DISTINCT 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 instructor.ID, instructor.name
FROM instructor
WHERE NOT EXISTS (
SELECT course.course_id
FROM course
WHERE course.dept_name = instructor.dept_name
EXCEPT
SELECT teaches.course_id
FROM teaches
WHERE teaches.ID = instructor.ID
)
ORDER BY instructor.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 student.ID, student.name
```

*FROM student*

*WHERE student.dept_name = 'History'*

*AND student.name LIKE 'D%'*

*AND (*

*SELECT COUNT(*)*

*FROM takes*

*JOIN course ON takes.course_id = course.course_id*

*WHERE course.dept_name = 'Music'*

*AND takes.ID = student.ID*

*) < 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. Indeed 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.**

*SELECT AVG(salary) - (SUM(salary) / COUNT(*)) FROM instructor;*

**Answer:** The issue arises if salary contains NULL values. AVG(salary) ignores NULL values, while SUM(salary) / COUNT(*) includes all rows (even those where salary is NULL), leading to a discrepancy.

Example:

| ID | Salary |
|---|---|
| 1 | 50000 |
| 2 | NULL |
| 3 | 70000 |

- AVG(salary) = (50000 + 70000) / 2 = 60000
- SUM(salary) / COUNT(*) = (50000 + 70000) / 3 = 40000
- The difference is 60000 - 40000 = 20000, not zero.

**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 DISTINCT instructor.ID, instructor.name*

*FROM instructor*

*WHERE instructor.ID NOT IN (*

*SELECT teaches.ID*

*FROM teaches*

*JOIN takes ON teaches.course_id = takes.course_id AND teaches.sec_id = takes.sec_id*

*WHERE takes.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 DISTINCT instructor.ID, instructor.name*

*FROM instructor*

*WHERE instructor.ID NOT IN (*

*SELECT teaches.ID*

*FROM teaches*

*JOIN takes ON teaches.course_id = takes.course_id AND teaches.sec_id = takes.sec_id*

*WHERE takes.grade = 'A'*

*)*

*AND instructor.ID IN (*

*SELECT teaches.ID*

*FROM teaches*

*JOIN takes ON teaches.course_id = takes.course_id AND teaches.sec_id = takes.sec_id*

*WHERE takes.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 afternoon hours (i.e., ends at or after 12:00). (You should eliminate duplicates if any.)**

*SELECT DISTINCT course.course_id, course.title*

*FROM course*

*JOIN section ON course.course_id = section.course_id*

*WHERE course.dept_name = 'Comp. Sci.'*

*AND section.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 “courseid, secid, year, semester, num”. You do not need to output sections with 0 students.**

*SELECT takes.course_id, takes.sec_id, takes.year, takes.semester, COUNT(*) AS num*

*FROM takes*

*GROUP BY takes.course_id, takes.sec_id, takes.year, takes.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 section_counts AS (*

*SELECT takes.course_id, takes.sec_id, takes.year, takes.semester, COUNT(*) AS num*

*FROM takes*

*GROUP BY takes.course_id, takes.sec_id, takes.year, takes.semester*

*),*

*max_count AS (*

*SELECT MAX(num) AS max_enrollment FROM section_counts*

*)*

*SELECT sc.course_id, sc.sec_id, sc.year, sc.semester, sc.num*

*FROM section_counts sc*

*JOIN max_count mc ON sc.num = mc.max_enrollment;*