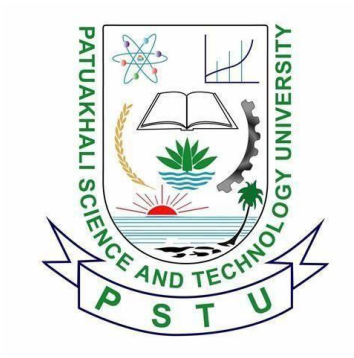

PATUAKHALI SCIENCE AND TECHNOLOGY UNIVERSITY

COURSE CODE CCE-224

SUBMITTED TO:

Prof. Dr. Md Samsuzzaman
Department of Computer and Communication Engineering
Faculty of Computer Science and Engineering

SUBMITTED BY:

Mohammed Sakib Hasan
ID: **2102052**,
Registration No: **10179**
Faculty of Computer Science and Engineering

Assignment 03
Assignment title: Chapter 03 (silberchatz)

Chapter 3 | Practice Exercises

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 website of the book, db-book.com. Instructions for setting up a database, and loading sample data, are provided on the above website.)

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 takes_table.ID from takes takes_table join teaches
teaches_table     on takes_table.course_id =
teaches_table.course_id     and takes_table.sec_id =
teaches_table.sec_id     and takes_table.semester =
teaches_table.semester     and takes_table.year =
teaches_table.year where teaches_table.ID = ( select ID from
instructor where name = "Einstein" ) ;
```

c. Find the highest salary of any instructor.

```
desc instructor;
select salary from
instructor order by
salary desc limit 1;
```

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

```
select ID, name from
instructor where
salary = (
      select salary
      from instructor
      order by salary desc
      limit 1
);
```

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

```
select course_id, sec_id, count(*)
from takes where year = 2017 and
semester = "Fall" group by
course_id, sec_id;
```

f. Find the maximum enrollment, a ross all se tions, in Fall 2017.

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

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

```
select sec_id, course_id
from takes where year =
2017 and semester = "Fall"
group by sec_id, course_id
having count(*) order by
count(*) desc limit 1;
```

2. Suppose you are given a relation grade points(grade, points) that provides a conversion 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.

```
select sum(credits * points) from
takes, course, grade_points where
takes.grade = grade_points.grade and
takes.course_id = course.course_id and
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(credits * points)/sum(credits) as GPA from
 takes, course, grade_points where
 takes.grade = grade_points.grade and
takes.course_id = course.course_id and
ID= '12345';
```

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

```
select
 ID, sum(credits * points)/sum(credits) as GPA from
 takes, ourse, grade_points where
 takes.grade = grade_points.grade and
takes.course_id = course.course_id
group by 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. Above solutions won‟t work if the answer is null. To fix it, we can use join operations or union operation with *null* filtering.

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
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 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 select ID,
name, dept_name, 100000 from
student where tot_cred > 100;
```

---

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

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 person.driver_id) from
 accident, participated, person, owns where
 accident.report_number = participated.report_number
and owns.driver_id = person.driver_id and
owns.license_plate = participated.license_plate and
year = 2017;
```

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

```
delete car where year = 2010 and
license_plate in
      (select license plate   from
owns o      where o.driver id =
 ‘12345’ )
```

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:

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

```
select ID,
      case
            when name=score < 40 then "F"
when name=score < 60 then "C"
when name=score < 80 then "B"               else
"A"      end from marks;
```

Find the number of students with each grade.

```
with grades as (
select ID,  case
            when name=score < 40 then "F"
        when name=score < 60 then "C"
when name=score < 80 then "B"
else "A"     end


from marks ) select grade, count(ID) from grades
group by grade;
```

6. **The SQL like operator is case sensitive (in most systems), but the lower() funtion 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%";
```

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. The query selects those values of p.a1 that are equal to some value of r1.a1 or r2.a1 if and only if both r1 and r2 are non-empty. If one or both of r1 and r2 are empty, the Cartesian product of p, r1 and r2 is empty, hen e the result of the query is empty. If p itself is empty, the result is empty.

8. Consider the bank database of Figure 3.18, where the primary keys are underlined. 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 ID
      from
depositor)  except
      (select ID
      from
       borrower)
```

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

```
select F.ID
from customer as F, customer as S where
       F.customer_street = S.customer_street
      and F.customer_city = S.customer_city
      and S.customer_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 branch name from account, depositor,
customer where  ustomer.id = depositor.id        and
depositor.account_number = account.account_number      and
customer_city = 'Harrison';
```

9. Consider the relational database of Figure 3.19, where the primary keys are underlined. Give an

employee (*ID*, *person_name*, *street*, *city*)
works (*ID*, *company_name*, *salary*)
company (*company_name*, *city*)
manages (*ID*, *manager_id*)

Figure 3.19 Employee database.

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.person name, city from employee as e,
works as w where w.company name =  "First Bank
Corporation"  and       w.ID = e.ID
```

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, city from employee as e, works as w
where w.company_name = "First Bank Corporation" and   w.salary
> 10000 and
      w.ID = e.ID;
```

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

```
      select 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”.

```
SELECT ID
FROM works
WHERE salary > (
    SELECT MAX(salary)
    FROM works
    WHERE 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.

```
SELECT S.company_name
FROM company AS S
WHERE NOT EXISTS (
    SELECT city
    FROM company
    WHERE company_name = 'Small Bank Corporation'
      AND city NOT IN (
          SELECT city
          FROM company AS T
          WHERE S.company_name = T.company_name
      )
);
```

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 (distinct ID) >= all   (select count
(distinct ID)     from works
      group by company_name)
```

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

```
SELECT company_name
FROM works
GROUP BY company_name
```

```
HAVING AVG(salary) > (
      SELECT AVG(salary)
      FROM works
      WHERE company_name = 'First Bank Corporation'
);
```

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”.

employee (*ID*, *person_name*, *street*, *city*)
works (*ID*, *company_name*, *salary*)
company (*company_name*, *city*)
manages (*ID*, *manager_id*)

Figure 3.19 Employee database.

```
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.

```
-- First update: Increase salary by 3% for managers earning more than 100,000 after a 10% raise
UPDATE works T
SET T.salary = T.salary * 1.03
WHERE T.ID IN (SELECT manager_id FROM manages)
  AND T.salary * 1.1 > 100000
  AND T.company_name = 'First Bank Corporation';

-- Second update: Increase salary by 10% for managers earning 100,000 or less after a 10% raise
UPDATE works T
SET T.salary = T.salary * 1.1
WHERE T.ID IN (SELECT manager_id FROM manages)
  AND T.salary * 1.1 <= 100000
  AND T.company_name = 'First Bank Corporation';
```

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 takes.ID
    FROM takes
    WHERE takes.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) AS lowest_max_salary
FROM (
    SELECT dept_name, MAX(salary) AS max_salary
    FROM instructor
    GROUP BY dept_name
) AS dept_max_salaries;
```

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)
VALUES ('CS-101', 1, 'Fall', 2017);
```

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

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

```
SELECT student.ID, 'CS-001', 1, 'Fall', 2017
FROM student
WHERE student.dept_name = 'Comp. Sci.';
```

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

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

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';
```

Note: If you run this delete statement without first deleting the sections of this course, it will fail due to foreign key constraints (if the database enforces referential integrity).

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%'
);
```

13.

Write SQL DDL corresponding to the schema in Figure 3.17. Make any reasonable assumptions about data

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

types, and be sure to declare primary and foreign keys.

```
-- Person Table CREATE TABLE person (
driver_id INT PRIMARY KEY,       name
VARCHAR(100) NOT NULL,        address
VARCHAR(100)
);

-- Car Table CREATE TABLE car (     license_plate
VARCHAR(20) PRIMARY KEY,      model VARCHAR(50),
year YEAR
);

-- Accident Table CREATE TABLE
accident (
    report_number INT PRIMARY KEY,      year
YEAR,
    location VARCHAR(100)
);

-- Owns Table
--  (Relationship between person and car)
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
);

-- Participated Table
-- (Records cars and drivers involved in accidents)
CREATE TABLE participated (
```

```
    report_number INT,
license_plate VARCHAR(20),
driver_id INT,
    damage_amount DECIMAL(10,2) DEFAULT 0.00,
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 SET NULL,
    FOREIGN KEY (driver_id) REFERENCES person(driver_id) ON DELETE SET NULL );
```

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 accident.report_number) AS num_accidents
FROM accident
JOIN participated ON accident.report_number = participated.report_number
JOIN person ON participated.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';
```

Consider the bank database of Figure 3.18, where the primary keys are underlined. 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 each customer who has an account at every branch located in “Brooklyn”.

```
SELECT customer_name
FROM customer C
WHERE NOT EXISTS (
    SELECT branch_name
```

15.

```
    FROM branch
    WHERE branch_city = 'Brooklyn'
    EXCEPT
    SELECT A.branch_name
    FROM account A
    JOIN depositor D ON A.account_number = D.account_number
    WHERE D.ID = C.ID
);
```

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 branch_name
FROM branch
WHERE assets > (
    SELECT MIN(assets)
    FROM branch
    WHERE branch_city = 'Brooklyn'
);
```

16.

Consider the employee database of Figure 3.19, where the primary keys are underlined. Give an expression in

employee (*ID*, *person_name*, *street*, *city*)
works (*ID*, *company_name*, *salary*)
company (*company_name*, *city*)
manages (*ID*, *manager_id*)

Figure 3.19 Employee database.

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 employee.ID, employee.person_name
FROM employee
JOIN works ON employee.ID = works.ID
JOIN company ON works.company_name = company.company_name
WHERE employee.city = company.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 ON e1.ID = manages.ID
JOIN employee e2 ON manages.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 employee.ID, employee.person_name
FROM employee
JOIN works ON employee.ID = works.ID
WHERE works.salary > (
SELECT AVG(salary)
    FROM works w2
    WHERE w2.company_name = works.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;
```

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

17.

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 manager_id FROM
manages);
```

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

```
DELETE FROM works
WHERE company_name = 'Small Bank Corporation';
```

---

employee (*ID*, *person_name*, *street*, *city*)
works (*ID*, *company_name*, *salary*)
company (*company_name*, *city*)
manages (*ID*, *manager_id*)

Figure 3.19 Employee database.

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) NOT NULL,      street
VARCHAR(100),      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) ON DELETE CASCADE,
    FOREIGN KEY (company_name) REFERENCES company(company_name)
);

CREATE TABLE manages (     ID INT,
manager_id INT,
    PRIMARY KEY (ID),
    FOREIGN KEY (ID) REFERENCES employee(ID) ON DELETE CASCADE,
    FOREIGN KEY (manager_id) REFERENCES employee(ID) ON DELETE SET NULL );
```

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

1. **Missing or Unknown Data**: A value may not be available or unknown at the time of data entry (e.g., a customer‟s middle name).
2. **Optional Attributes**: Some attributes may not apply to all entities (e.g., a `end_date` for an employee who is still employed).

1. Show that, in SQL, <> ALL is identical to NOT IN.

```
-- Using <> ALL
SELECT *
FROM table
WHERE column <> ALL (SELECT value FROM other_table);
-- Using NOT IN
SELECT *
FROM table
WHERE column NOT IN (SELECT value FROM other_table);
```

Both queries return the same result: rows where `column` does not match any value in the subquery.

2. Consider the library database of Figure 3.20. Write

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

Figure 3.20 Library database.

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 m.memb_no, m.name
FROM member m
JOIN borrowed b ON m.memb_no = b.memb_no
JOIN book bk ON b.isbn = bk.isbn WHERE bk.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 bk.isbn
    FROM book bk
    WHERE bk.publisher = 'McGraw-Hill'
    EXCEPT
    SELECT b.isbn
    FROM borrowed b
    WHERE b.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 bk.publisher, m.memb_no, m.name FROM member m
JOIN borrowed b ON m.memb_no = b.memb_no JOIN book bk ON b.isbn =
bk.isbn
GROUP BY bk.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_count) AS avg_books_borrowed
FROM (
    SELECT m.memb_no, COUNT(b.isbn) AS borrowed_count
    FROM member m
    LEFT JOIN borrowed b ON m.memb_no = b.memb_no
    GROUP BY m.memb_no
) AS member_borrow_counts;
```

3. Rewrite the `where` clause without using the `unique` construct.

```
where unique (select title from course)
```

```
-- We can use a equivalent test with distinct value and all values instead of where
clause

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

4. Rewrite the following query without using the `with` construct.

Original 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;
```

Rewritten Query:

```
SELECT dept_name
FROM (
    SELECT dept_name, SUM(salary) AS total_salary
    FROM instructor
    GROUP BY dept_name
) AS dept_total
WHERE dept_total.total_salary >= (
    SELECT AVG(total_salary)
    FROM (
        SELECT SUM(salary) AS total_salary
        FROM instructor
        GROUP BY dept_name
    ) AS avg_salaries
);
```

5. 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.name, s.ID
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';
```

6. 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');
```

7. 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 course_id, ID
from takes
    group by ID, course_id having count(*) > 2
order by course_id;
```

8. 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
WHERE t.grade IS NOT NULL AND 2 <=
(     select count(course_id)
from takes t2     where t2.ID =
t.ID     and t2.course_id =
t.course_id
)
GROUP BY t.ID
HAVING COUNT(DISTINCT t.course_id) >= 3;
```

9. 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 the result by name.

```
SELECT i.name, i.ID
FROM instructor i
WHERE NOT EXISTS (
    SELECT c.course_id
    FROM course c
    WHERE c.dept_name = i.dept_name
    EXCEPT
    SELECT t.course_id
    FROM teaches t
    WHERE t.ID = i.ID
)
ORDER BY i.name;
```

10. 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
    WHERE t.ID = s.ID
    AND t.course_id IN (
        SELECT c.course_id
        FROM course c
        WHERE c.dept_name = 'Music'
    )
) < 5;
```

11. 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.

The result might not be zero if there are instructors with missing or null salary data. The `AVG()` function ignores nulls, while `SUM(salary)` and `COUNT(*)` count all rows, potentially leading to discrepancies.

12. 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
LEFT JOIN teaches t ON i.ID = t.ID
LEFT JOIN takes tk ON t.course_id = tk.course_id
WHERE tk.grade != 'A' OR tk.grade IS NULL
GROUP BY i.ID;
```

13. 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 i.ID, i.name
FROM instructor i
LEFT JOIN teaches t ON i.ID = t.ID
LEFT JOIN takes tk ON t.course_id = tk.course_id
WHERE tk.grade != 'A' AND EXISTS (  select tk2.grade
from takes tk2     where t.course_id = tk2.course_id
and tk2.grade is not null
);
```

14. 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 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 t.end_hr >= 12 AND c.dept_name = 'Comp. Sci.';
```

15. 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 s.course_id, s.sec_id, s.year, s.semester,
COUNT(t.ID) AS num
FROM section s
LEFT JOIN takes t ON s.course_id = t.course_id
AND s.sec_id = t.sec_id
GROUP BY course_id, sec_id, year, semester
HAVING num > 0;
```

16. 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_enrollment AS (
    SELECT s.course_id, s.sec_id, s.year, s.semester, COUNT(t.ID) AS num
FROM section s
    LEFT JOIN takes t ON s.course_id = t.course_id
    AND s.sec_id = t.sec_id
    GROUP BY course_id, sec_id, year, semester )
SELECT course_id, sec_id, year, semester, num
FROM section_enrollment
WHERE num = (SELECT MAX(num) FROM section_enrollment);
```