> *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.*
> *b. Find the IDs of all students who were taught by an instructor named Einstein; make sure there are no duplicates in the result.*
> *c. Find the highest salary of any instructor.*
> *d. Find all instructors earning the highest salary (there may be more than one with the same salary).*
> *e. Find the enrollment of each section that was offered in Fall 2017.*
> *f. Find the maxium enrollment, across all sections, in Fall 2017.*
> *g. Find the sections that had the maximum enrollment in Fall 2017.*

---

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

```sql
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. This query can be answered in several different ways. One way is as follows.

```sql
SELECT DISTINCT takes.ID
FROM takes, instructor, teaches
WHERE takes.course_id = teaches.course_id AND
    takes.sec_id = teaches.sec_id AND
    takes.semester = teaches.semester AND
    takes.year = teaches.year AND
    teaches.id = instructor.id AND
    instructor.name = 'Einstein'
```

Another method by using subqueries

```sql
SELECT DISTINCT id
FROM takes
WHERE (course_id, sec_id, semester, year) IN
(
    SELECT course_id, sec_id, semester, year
    FROM teaches INNER JOIN instructor
        ON teaches.id = instructor.id
    WHERE instructor.name = 'Einstein'
);
```

c. Find the highest salary of any instructor.

```sql
SELECT MAX(salary)
FROM instructor
```

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 MAX(salary) FROM instructor)
```

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

```
SELECT course_id, sec_id, (
    SELECT COUNT(id)
    FROM takes
    WHERE takes.year = section.year
        AND takes.semester = section.semester
        AND takes.course_id = section.course_id
        AND takes.sec_id = section.sec_id
) AS enrollment
FROM section
WHERE semester = 'Fall' AND year = 2017
```

Note that if the result of the subquery is empty, the aggregate function **count** returns a value of 0. One way of writing the query might appear to be:

```
SELECT course_id, sec_id, COUNT(id)
FROM takes
WHERE semester = 'Fall' AND year = 2017
GROUP BY course_id, sec_id
```

But note that if a section does not have any students taking it, it would not appear in the result. One way of ensuring such a section appears with a count of 0 is to use the **outer join** operation, covered in Chapter 4.

f. Find the maxium enrollment, across all sections, in Fall 2017. One way of writing this query is as follows:

```
WITH enrollment_in_fall_2017(course_id, sec_id, enrollment) AS (
    SELECT course_id, sec_id, COUNT(id)
    FROM takes
    WHERE semester = 'Fall' AND year = 2017
    GROUP BY course_id, sec_id
)
SELECT CASE
        WHEN MAX(enrollment) IS NOT NULL THEN MAX(enrollment)
        ELSE 0
       END
FROM enrollment_in_fall_2017;
```

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

```
WITH enrollment_in_fall_2017(course_id, sec_id, enrollment) AS (
    SELECT course_id, sec_id, COUNT(id)
    FROM takes
    WHERE semester = 'Fall' AND year = 2017
    GROUP BY course_id, sec_id
)
SELECT course_id, sec_id
FROM enrollment_in_fall_2017
WHERE enrollment = (SELECT MAX(enrollment) FROM enrollment_in_fall_2017);
```

It is also possible to write the query without the **with** clause, but the subquery to find enrollment would get repeated twice in the query. While not incorrect to add **distinct** in the **count**, it is not necessary in light of the primary key constraint on *takes*. > 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.*
> *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.*

---

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

```
SELECT SUM(c.credits * g.points)
FROM takes AS t
    INNER JOIN course AS c ON t.course_id = c.course_id
    INNER JOIN grade_points AS g ON t.grade = g.grade
WHERE t.ID = '12345'
```

In the above query, a student who has not taken any course would not have any tuples, whereas we would expect to get 0 as the answer. One way of fixing this problem is to use the **outer join** operation, which we study later in Chapter 4. Another way to ensure that we get 0 as the answer is via the following query:

```
(
SELECT SUM(c.credits * g.points)
FROM takes AS t
    INNER JOIN course AS c ON t.course_id = c.course_id
    INNER JOIN grade_points AS g ON t.grade = g.grade
WHERE t.ID = '12345'
)
UNION
(
SELECT 0
FROM student
WHERE ID = '12345' AND NOT EXISTS (SELECT * FROM takes WHERE 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(c.credits * g.points) / SUM(credits) AS GPA
FROM takes AS t
    INNER JOIN course AS c ON t.course_id = c.course_id
    INNER JOIN grade_points AS g ON t.grade = g.grade
WHERE t.ID = '12345'
```

As before, a student who has not taken any course would not appearn in the above result; we can ensure that such a stuent appears in the result by using the modified query from the previous part of this question. However, an additional issue in this case is that the sum of credits would also be 0, resulting in a divide-by-zero condition. In fact, the only meaningful way of defining the GPA in this case is to define it as *null*. We can ensure that such a student appears in the result with a null *GPA* by adding the following **union** clause to the above query.

```
UNION
(
    SELECT null AS GPA
    FROM student
    WHERE ID = '12345' AND
        NOT EXISTS (SELECT * FROM takes WHERE ID='12345')
)
```

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

```
SELECT t.ID, SUM(c.credits * g.points) / SUM(credits) AS GPA
FROM takes AS t
    INNER JOIN course AS c ON t.course_id = c.course_id
    INNER JOIN grade_points AS g ON t.grade = g.grade
GROUP BY t.ID
```

Again, to handle students who have not taken any course, we would have to add the following **union** clause:

```
UNION
(
    SELECT ID, null AS GPA
    FROM student
    WHERE NOT EXISTS (SELECT * FROM takes WHERE takes.ID = student.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.

The queries listed above all include a test of equality on *grade* between *grade_points* and *takes*. Thus, for any *takes* tuple with a *null* grade, that student's course would be eliminated from the rest of the computation of the result. As a result, the credits of such courses would be eliminated also, and thus the queries would return the correct answer even if some grades are null.

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

> *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.*

---

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

```sql
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).

```sql
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.

~~~sql
INSERT INTO instructor
SELECT ID, name, dept_name, 10000
FROM student
WHERE tot_cred > 100
```> Consider the insurance database of Figure 3.17, where the primary keys
> are underlined. Construct the following SQL queries for this relational database.
<br>
> a. Find the total number of people who owned cars that were involved in accidents
in 2017. <br>
> b. Delete all year-2010 cars belonging to the person whose ID is '12345'. <br>

--------------------------------

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

Note: This is not the same as the total number of accidents in 2017. We must count
people
with several accidents only once. Furthermore, note that the question asks for
owners,
and it might be that the owner of the car was not the driver actually involved in
the
accident.

```sql
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 FROM car
WHERE year = 2010 AND license_plate IN
    (SELECT license_plate FROM owns WHERE driver_id = '12345')
```

Note: The *owns*, *accident* and *participated* records associated with the deleted cars still exist. (If you want the corresponding rows in *owns* and *participated* to also be deleted you can use **ON CASCADE DELETE** on the schema of the tables).> 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 $\leq$ score $<$ 60, grade B if 60 $\leq$ score $<$ 80, and grade A if 80 $\leq$ score. Write SQL queries to do the following:*
>
> *a. Display the grade for each student, based on the marks relation.*
> *b. Find the number of students with each grade.*

---

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

```
SELECT ID,
    CASE
        WHEN score < 40 THEN 'F'
        WHEN score < 60 THEN 'C'
        WHEN score < 80 THEN 'B'
        ELSE 'A'
    END
FROM marks
```

b. Find the number of students with each grade.

```
WITH grades(ID,grade) AS (
    SELECT ID,
        CASE
            WHEN score < 40 THEN 'F'
            WHEN score < 60 THEN 'C'
            WHEN score < 80 THEN 'B'
            ELSE 'A'
        END
    FROM marks
)
SELECT grade, COUNT(ID)
FROM grades
GROUP BY grade
```

As an alternative, the **WITH** clause can be removed, and instead the definition of *grades* can be made a subquery of the main query. > 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%'
```> Consider the SQL query
```sql
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, hence the result of the query is empty. If *p* itself is empty, the result is empty. > 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.*
> *b. Find the ID of each customer who lives on the same street and in the same city as customer '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".*

---

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';
```

Another method (using **scalar subqueries**)

```
SELECT ID
FROM customer
WHERE customer_street = (SELECT customer_street FROM customer WHERE ID = '12345')
AND
    customer_city = (SELECT customer_city FROM customer WHERE 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 customer.id = depositor.id
    AND depositor.account_number = account.account_number
    AND customer_city = 'Harrison'
```> 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". <br>
> b. Find the ID, name, and city of residence of each employee who works for "First
Bank Corporation"
> and earns more than $10000. <br>
> c. Find the ID of each employee who does not work for "First Bank Corporation".
<br>
> d. Find the ID of each employee who earns more than every employee of "Small Bank
Corporation". <br>
> 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. <br>
> 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). <br>
> g. Find the name of each company whose employees earn a higher salary, on average,
than the
> average salary at "First Bank Corporation". <br>

--------------------------------



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

```sql
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 ID, name, city
FROM employee
WHERE ID IN (
    SELECT ID
    FROM works
    WHERE company_name = 'First Bank Corporation' AND salary > 10000
)
```

This could be written also in the style of the answer to part a, as follows:

```
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
    AND w.salary > 10000
```

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'
```

If one allows people to appear in *employee* without appearing also in *works*, the solution is slightly more complicated. An outer join as discussed in Chapter 4 could be used as well.

```
SELECT ID
FROM employee
WHERE ID NOT IN (
    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 > ALL (
    SELECT salary
    FROM works
    WHERE company_name = 'Small Bank Corporation'
)
```

If people may work for several companies and we wish to consider the *total* earnings of each person, the is more complex. But note that the fact that ID is the primary key for *works* implies that this cannot be the case.

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'
    )
    EXCEPT
    (
        SELECT city
        FROM company AS T
        WHERE T.company_name = S.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'
)
```

> *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".*
> *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.*

---

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 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';
```

```
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';
```

The above updates would give different results if executed in the opposite order. We give below a safer solution using the **case** statement.

```
UPDATE works T
SET T.salary = T.salary * (
    CASE
        WHEN (T.salary * 1.1 > 100000) THEN 1.03
        ELSE 1.1
    END
)
WHERE T.ID IN (SELECT manager_id FROM manages)
    AND T.company_name = 'First Bank Corporation'
```

> *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 duplicates names in the result.*
> *b. Find the ID and name of each student who has not taken any course offered before 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.*
> *d. Find the lowest, across all departments, of the per-department maximum salary computed by the preceding query.*

---

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

```
SELECT DISTINCT student.ID, student.name
FROM student INNER JOIN takes  ON student.ID = takes.ID
             INNER 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 ID, name
FROM student AS S
WHERE NOT EXISTS (
    SELECT *
    FROM takes AS T
    WHERE year < 2017 AND T.ID = S.ID
)
```

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)
FROM instructor
GROUP BY dept_name
```

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

```
WITH maximum_salary_within_dept(dept_name, max_salary) AS (
    SELECT dept_name, MAX(salary)
    FROM instructor
    GROUP BY dept_name
)
SELECT MIN(max_salary)
FROM maximum_salary_within_dept
```

> *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.*
> *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.*
> *c. Enroll every student in the Comp. Sci. department in the above section.*
> *d. Delete enrollments in the above section where the student's ID is 12345.*
> *e. Delete the course CS-001. What will happen if you run this **delete** statement without first deleting offerings (sections) of this course?*
> *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.*

---

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-001', '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 ID = '12345' AND (course_id, sec_id, semester, year) = ('CS-001', '1', 'Fall',
2017)
```

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

To delete the course CS-001 we use the following query:

```
DELETE FROM course
WHERE course_id = 'CS-001';
```

Below we can see the schema of the *section* relation. Taken from db-book.com.

```
create table section
        (course_id              varchar(8),
         sec_id                 varchar(8),
         semester               varchar(6)
                check (semester in ('Fall', 'Winter', 'Spring', 'Summer')),
         year                   numeric(4,0) check (year > 1701 and year < 2100),
         building               varchar(15),
         room_number            varchar(7),
         time_slot_id           varchar(4),
         primary key (course_id, sec_id, semester, year),
         foreign key (course_id) references course (course_id)
                on delete cascade,
         foreign key (building, room_number) references classroom (building,
room_number)
                on delete set null
        );
```

In the above query we see **ON DELETE CASCADE**. This means when any *course* tuple is deleted all corresponding *section* tuples will also be deleted. Therefore when you delete CS-001 all the corresponding rows in the *section* relation will also be deleted.

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%'
)
```

> *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 VARCHAR(15),
    name VARCHAR(30) NOT NULL,
    address VARCHAR(40),
    PRIMARY KEY (driver_id)
```

```sql
);

CREATE TABLE car (
    license_plate VARCHAR(8),
    model VARCHAR(7),
    year NUMERIC(4,0) CHECK (year > 1701 AND year < 2100),
    PRIMARY KEY (license_plate)
);

CREATE TABLE accident (
    report_number VARCHAR(10),
    year NUMERIC(4,0) CHECK (year > 1701 AND year < 2100),
    location VARCHAR(30),
    PRIMARY KEY (report_number)
);

CREATE TABLE owns (
    driver_id VARCHAR(15),
    license_plate VARCHAR(8),
    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 VARCHAR(10),
    license_plate VARCHAR(8),
    driver_id VARCHAR(15),
    damage_amount NUMERIC(10,2),
    PRIMARY KEY(report_number, license_plate),
    FOREIGN KEY (report_number) REFERENCES accident(report_number),
    FOREIGN KEY (license_plate) REFERENCES car(license_plate)
);
```

> *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".*
> *b. Update the damage amount for the car with license_plate "AABB2000" in the accident with report number "AR2197" to $3000.*

---

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

```sql
WITH all_cars_owned_by_john_smith(license_plate) AS (
    SELECT license_plate
    FROM person INNER JOIN owns ON person.driver_id = owns.driver_id
    WHERE person.name = 'John Smith'
)
SELECT COUNT(DISTINCT report_number)
```

```
FROM participated
WHERE license_plate IN (SELECT license_plate FROM all_cars_owned_by_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.
>
> a. Find each customer who has an account at _every_ branch located in "Brooklyn".
<br>
> b. Find the total sum of all loan amounts in the bank. <br>
> c. Find the names of all branches that have assets greater than those
> of at least one branch located in "Brooklyn" <br>

--------------------------------

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

```sql
WITH all_branches_in_brooklyn(branch_name) AS (
    SELECT branch_name
    FROM branch
    WHERE branch_city = 'Brooklyn'
)
SELECT ID, customer_name
FROM customer AS c1
WHERE NOT EXISTS (
    (SELECT branch_name FROM all_branches_in_brooklyn)
    EXCEPT
    (
        SELECT branch_name
        FROM account INNER JOIN depositor
            ON account.account_number = depositor.account_number
        WHERE depositor.ID = c1.ID
    )
)
~~~

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

```
SELECT SUM(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 > SOME (
    SELECT assets
    FROM branch
    WHERE branch_city = 'Brooklyn'
);
```> Consider the employee database of Figure 3.19, where the primary keys are
underlined.
> Given 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. <br>
> 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. <br>
> c. Find ID and name of each employee who earns more than the average salary of
> all employees of her or his company. <br>
> d. Find the company that has the smallest payroll. <br>

--------------------------------

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.

```sql
SELECT employee.id, employee.person_name
FROM employee INNER JOIN works ON employee.id = works.id
              INNER 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 E.id, E.person_name
FROM employee AS E INNER JOIN manages ON E.id = manages.id
                   INNER JOIN employee AS manager_of_E ON manages.manager_id =
manager_of_E.id
WHERE E.street = manager_of_E.street AND
      E.city = manager_of_E.city;
````

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

````
WITH average_salary_per_company(company_name, avg_salary) AS (
    SELECT company_name, AVG(salary)
    FROM works
    GROUP BY company_name
)
SELECT E.id, E.person_name
FROM employee AS E INNER JOIN works ON E.id = works.id
WHERE works.salary > (
````

```
    SELECT avg_salary
    FROM average_salary_per_company
    WHERE company_name = works.company_name
)
```

d. Find the company that has the smallest payroll.

```sql
SELECT company_name, SUM(salary) AS total_payroll
FROM works
GROUP BY company_name
ORDER BY total_payroll ASC
LIMIT 1
```

> *Consider the employee database of Figure 3.19. Give an expression in SQL for each of the following queries.*
>
> *a. Give all employees for "First Bank Corporation" a 10 percent raise.*
> *b. Give all managers of "First Bank Corporation" a 10 percent raise.*
> *c. Delete all tuples in the works relation for employees of "Small Bank Corporation".*

---

a. Give all employees for "First Bank Corporation" a 10 percent raise.

```sql
UPDATE works
SET salary = salary * 1.1
WHERE company_name = 'First Bank Corporation'
```

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

```sql
UPDATE works
SET salary = salary * 1.1
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".

~~~sql
DELETE FROM works
WHERE company_name = 'Small Bank Corporation'
```> 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.

--------------------------------


```sql
CREATE TABLE employee (
    id VARCHAR(8),
    person_name VARCHAR(30) NOT NULL,
    street VARCHAR(40),
~~~

~~~~
    city VARCHAR(30),
    PRIMARY KEY (id)
);

CREATE TABLE company (
    company_name VARCHAR(40),
    city VARCHAR(30),
    PRIMARY KEY (company_name)
);

CREATE TABLE works (
    id VARCHAR(8),
    company_name VARCHAR(40),
    salary NUMERIC(10,2) CHECK (salary > 10000),
    PRIMARY KEY (id),
    FOREIGN KEY (id) REFERENCES employee(id)
        ON DELETE CASCADE,
    FOREIGN KEY (company_name) REFERENCES company(company_name)
        ON DELETE CASCADE
);

CREATE TABLE manages (
    id VARCHAR(8),
    manager_id VARCHAR(8),
    PRIMARY KEY (id),
    FOREIGN KEY (id) REFERENCES employee (id),
    FOREIGN KEY (manager_id) REFERENCES employee (id)
);
```> List two reasons why null values might be introduced into the database.

--------------------------------

[same question as that of 2.16]
1. When a value of an attribute is unknown.
2. When a value of an attribute does not exist

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

--------------------------------

Let _r1_ and _r2_ be two relations having only one attribute say _k_.

Let the following be called "query 1".

```sql
SELECT k
FROM r1
WHERE k <> ALL (
    SELECT k
    FROM r2
)
~~~~

Let the following be called "query 2".

```
SELECT k
FROM r1
WHERE k NOT IN (
    SELECT k
    FROM r2
)
```

As you can see *query 1* and *query 2* are almost the same, except we replaced "<> ALL" in *query 1* by "NOT IN" in *query 2*. If we can show that the two queries give the same result i.e. the same set of tuples, we have shown that **$<>$ ALL** is identical to **NOT IN**.

Take any tuple from the result of *query 1*, say t_1. Since t_1 is in relation $r1$ and not equal to any tuple of relation $r2$, it is not in relation $r2$. Therefore t_1 is in the result of *query 2*.

Take any tuple from the result of *query 2*, say t_2. Since t_2 is in relation $r1$ and not in relation $r2$, it is not equal to any tuple of relation $r2$. Therefore t_2 is in the result of *query 1*.

Thus the result of the two queries is identical.

Which proves in general case that **$<>$ ALL** is identical to **NOT IN**.> 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".*
> *b. Find the member number and name of each member who has borrowed every book published by "McGraw-Hill".*
> *c. For each publisher, find the member number and name of each member who has borrowed more than five books of that publisher.*
> *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.*

---

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

```
SELECT memb_no, name
FROM member AS m
WHERE EXISTS (
    SELECT *
    FROM book INNER JOIN borrowed ON book.isbn = borrowed.isbn
    WHERE book.publisher = 'McGraw-Hill' AND
            borrowed.memb_no = m.memb_no
)
```

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

```sql
SELECT memb_no, name
FROM member AS m
WHERE NOT EXISTS (
    (
        SELECT isbn
        FROM book
        WHERE publisher = 'McGraw-Hill'
    )
    EXCEPT
    (
        SELECT isbn
        FROM borrowed
        WHERE 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.

```sql
WITH member_borrowed_book(memb_no, memb_name,isbn,title,authors,publisher,date) AS (
    SELECT member.memb_no, name, book.isbn, title, authors, publisher, date
    FROM member INNER JOIN borrowed ON member.memb_no = borrowed.memb_no
                INNER JOIN book ON borrowed.isbn = book.isbn
)
SELECT memb_no, memb_name, publisher, COUNT(isbn)
FROM member_borrowed_book
GROUP BY memb_no, memb_name, publisher
HAVING COUNT(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.

~~~sql
WITH number_of_books_borrowed(memb_no, memb_name, number_of_books) AS (
    SELECT memb_no, name, (
        CASE
            WHEN NOT EXISTS (SELECT * FROM borrowed WHERE borrowed.memb_no =
member.memb_no) THEN 0
            ELSE (SELECT COUNT(*) FROM borrowed WHERE borrowed.memb_no =
member.memb_no)
        END
    )
    FROM member
)
SELECT AVG(number_of_books) AS average_number_of_books_borrowed_per_member
FROM number_of_books_borrowed
```> Rewrite the **WHERE** clause
```sql
WHERE UNIQUE (SELECT title FROM course)
~~~

> *without using the **UNIQUE** construct.*

One method

```
WHERE 1 >= ALL (
    SELECT COUNT(*)
    FROM course
    GROUP BY title
)
```

Another method

~~~
WHERE NOT EXISTS (
    SELECT *
    FROM course AS c1, course AS c2
    WHERE c1.course_id != c2.course_id AND c1.title = c2.title
)
```> Consider the query:

```sql
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 x) AS dept_total_avg
WHERE dept_total.value >= dept_total_avg.value
```> 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.

--------------------------------

```sql
SELECT s.id, s.name
FROM student AS s INNER JOIN advisor AS a ON s.id = a.s_id
                  INNER JOIN instructor AS i ON a.i_id = i.id
WHERE s.dept_name = 'Accounting' AND i.dept_name = 'Physics';
~~~

~~~~
```> 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.

--------------------------------

```sql
SELECT dept_name
FROM department
WHERE budget > (SELECT budget FROM department WHERE dept_name = 'Philosophy')
ORDER BY dept_name ASC;
```> 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.

--------------------------------

```sql
SELECT id,course_id
FROM takes
GROUP BY id,course_id
HAVING COUNT(*) >= 3
ORDER BY course_id ASC;
```> 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).

--------------------------------

```sql
WITH retakers(id,course_id,frequency) AS (
    SELECT id,course_id,COUNT(*)
    FROM takes
    GROUP BY id,course_id
    HAVING COUNT(*) > 1
)
SELECT id
FROM retakers
GROUP BY id
HAVING COUNT(*) >= 3;
```> 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.

--------------------------------
~~~~

~~~
```sql
SELECT id, name
FROM instructor AS i
WHERE NOT EXISTS (
    (SELECT course_id FROM course WHERE dept_name = i.dept_name)
    EXCEPT
    (SELECT course_id FROM teaches WHERE teaches.id = i.id)
)
ORDER BY name ASC
```> 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.

--------------------------------

```sql
SELECT id,name
FROM student AS s
WHERE dept_name = 'History'
    AND name LIKE 'D%'
    AND (
        SELECT COUNT(DISTINCT course_id)
        FROM takes
        WHERE takes.id = s.id AND
            course_id IN (SELECT course_id FROM course WHERE dept_name = 'Music')
    ) < 5;
```> Consider the following SQL query on the university schema:

```sql
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.*

Consider the following instance of the *instructor* relation.

```
university=# SELECT * FROM instructor;
  id   |    name    | dept_name  |  salary
-------+------------+------------+-----------
 12121 | Wu         | Finance    |  90000.00
 15151 | Mozart     | Music      |  40000.00
 1     | Tesla      | Elec. Eng. |   NULL
(3 rows)
```

Then using the previous query will not give 0 as shown below:-

```
university=# SELECT AVG(salary) - (SUM(salary)/COUNT(*))
university-# FROM instructor
university-# ;
      ?column?
--------------------
 21666.666666666667
(1 row)

university=#
```

This is because of the NULL salary of the instructor 'Tesla'.

All aggregate functions except **count(*)** ignore null values in their input collection.

Thus *AVG(salary)* computes:

$$ \frac {90000 + 40000} {2} = \frac {130000} {2} = 65000.0 $$

While *SUM(salary)/COUNT(*)* computes:

$$ \frac {SUM(salary)}{COUNT(*)} = \frac {90000 + 40000} {3} = \frac {130000} {3} = 43333.333333333336 $$

Which are not equal. Which explains the non-zero answer of the query in question. > 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 id, name
FROM instructor AS i
WHERE 'A' NOT IN (
    SELECT takes.grade
    FROM takes INNER JOIN teaches
        ON (takes.course_id,takes.sec_id,takes.semester,takes.year) =
           (teaches.course_id,teaches.sec_id,teaches.semester,teaches.year)
    WHERE teaches.id = i.id
)
```> 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.

--------------------------------

So the question paraphrased is as follows: 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 and who has given at least one other non-null grade in some course.


```sql
SELECT id, name
FROM instructor AS i
WHERE 'A' NOT IN (
~~~

~~~
    SELECT takes.grade
    FROM takes INNER JOIN teaches
        ON (takes.course_id,takes.sec_id,takes.semester,takes.year) =
           (teaches.course_id,teaches.sec_id,teaches.semester,teaches.year)
    WHERE teaches.id = i.id
)
AND
(
    SELECT COUNT(*)
    FROM takes INNER JOIN teaches
        ON (takes.course_id,takes.sec_id,takes.semester,takes.year) =
           (teaches.course_id,teaches.sec_id,teaches.semester,teaches.year)
    WHERE teaches.id = i.id AND takes.grade IS NOT NULL
) >= 1
```> 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.)

--------------------------------

```sql
SELECT course_id,title
FROM course AS c
WHERE dept_name = 'Comp. Sci.' AND
    EXISTS (
        SELECT *
        FROM section
        WHERE section.course_id = c.course_id AND
        time_slot_id IN (SELECT time_slot_id FROM time_slot WHERE end_hr >= 12)
    )
```> 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.

--------------------------------

```sql
SELECT course_id, sec_id, year, semester, COUNT(DISTINCT ID) AS num
FROM takes
GROUP BY course_id, sec_id, year, semester;
```> 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.)

--------------------------------

```sql
WITH section_student_frequency(courseid, secid, year, semester, num) AS (
    SELECT course_id, sec_id, year, semester, COUNT(DISTINCT ID)
    FROM takes
    GROUP BY course_id, sec_id, year, semester
)
~~~

```
SELECT *
FROM section_student_frequency
WHERE num = (SELECT MAX(num) FROM section_student_frequency);
```
```
```