CHAPTER 3

Introduction to SQL

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

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

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 Autumn 2009.

f. Find the maximum enrollment, across all sections, in Autumn 2009.

g. Find the sections that had the maximum enrollment in Autumn 2009.

Answer:

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

```
select   distinct student.ID
from     (student join takes using(ID))
         join (instructor join teaches using(ID))
         using(course_id, sec_id, semester, year)
where    instructor.name = 'Einstein'
```

As an alternative to th **join .. using** syntax above the query can be written by enumerating relations in the **from** clause, and adding the corresponding join predicates on *ID*, *course_id*, *section_id*, *semester*, and *year* to the **where** clause.
Note that using natural join in place of **join .. using** would result in equating student *ID* with instructor *ID*, which is incorrect.

c. Find the highest salary of any instructor.

```
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 Autumn 2009.
One way of writing the query is as follows.

```
select     course_id, sec_id, count(ID)
from       section natural join takes
where      semester = 'Autumn'
and        year = 2009
group by course_id, sec_id
```

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 replace **natural join** by the **natural left outer join** operation, covered later in Chapter 4. Another way is to use a subquery in the **select** clause, as follows.

```
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.section_id = section.section_id)
        from   section
where   semester = 'Autumn'
and     year = 2009
```

Note that if the result of the subquery is empty, the aggregate function **count** returns a value of 0.

f. Find the maximum enrollment, across all sections, in Autumn 2009. One way of writing this query is as follows:

```
select  max(enrollment)
from    (select    count(ID) as enrollment
         from      section natural join takes
         where     semester = 'Autumn'
         and       year = 2009
         group by course_id, sec_id)
```

As an alternative to using a nested subquery in the **from** clause, it is possible to use a **with** clause, as illustrated in the answer to the next part of this question.
A subtle issue in the above query is that if no section had any enrollment, the answer would be empty, not 0. We can use the alternative using a subquery, from the previous part of this question, to ensure the count is 0 in this case.

g. Find the sections that had the maximum enrollment in Autumn 2009. The following answer uses a **with** clause to create a temporary view, simplifying the query.

```
with  sec_enrollment as (
        select    course_id, sec_id, count(ID) as enrollment
        from      section natural join takes
        where     semester = 'Autumn'
        and       year = 2009
        group by course_id, sec_id)
select  course_id, sec_id
from    sec_enrollment
where   enrollment = (select max(enrollment) from sec_enrollment)
```

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.

3.2 Suppose you are given a relation *grade_points*(*grade*, *points*), which 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 above relation, and our university schema, write each of the following queries in SQL. You can 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 every student.

Answer:

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 natural join course) natural join grade_points
whereID = '12345'
```

One problem with the above query is that if the student has not taken any course, the result 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 **natural left outer join** operation, which we study later in Chapter 4. Another way to ensure that we get 0 as the answer, is to the following query:

```
(select   sum(credits * points)
from      (takes natural join course) natural join grade_points
where     ID = '12345')
union
(select   0
from      student
where     takes.ID = '12345' and
          not exists ( select * from takes where takes.ID = '12345'))
```

As usual, specifying join conditions can be specified in the **where** clause instead of using the **natural join** operation or the **join .. using** operation.

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 natural join course) natural join grade_points
where   ID = '12345'
```

As before, a student who has not taken any course would not appear in the above result; we can ensure that such a student 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    takes.ID = '12345' and
         not exists ( select * from takes where takes.ID = '12345'))
```

Other ways of ensuring the above are discussed later in the solution to Exercise 4.5.

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

```
select    ID, sum(credits * points)/sum(credits) as GPA
from      (takes natural join course) natural join grade_points
group by 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))
```

3.3

3.4 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 (that is, 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.

Answer:

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 (that is, 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, 10000
from   student
where  tot_cred > 100
```

3.5 Consider the insurance database of Figure **??**, 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 1989.

b. Add a new accident to the database; assume any values for required attributes.

c. Delete the Mazda belonging to "John Smith".

Answer: Note: The *participated* relation relates drivers, cars, and accidents.

a. Find the total number of people who owned cars that were involved in accidents in 1989.
Note: this is not the same as the total number of accidents in 1989. We must count people with several accidents only once.

```
select  count (distinct name)
from    accident, participated, person
where   accident.report_number = participated.report_number
and     participated.driver_id = person.driver_id
and     date between date '1989-00-00' and date '1989-12-31'
```

person (*driver_id*, *name*, *address*)
car (*license*, *model*, *year*)
accident (*report_number*, *date*, *location*)
owns (*driver_id*, *license*)
participated (*driver_id*, *car*, *report_number*, *damage_amount*)

Figure ??. Insurance database.

b. Add a new accident to the database; assume any values for required attributes.
We assume the driver was "Jones," although it could be someone else. Also, we assume "Jones" owns one Toyota. First we must find the license of the given car. Then the *participated* and *accident* relations must be updated in order to both record the accident and tie it to the given car. We assume values "Berkeley" for *location*, '2001-09-01' for date and *date*, 4007 for *report_number* and 3000 for damage amount.

```
insert into accident
        values (4007, '2001-09-01', 'Berkeley')

insert into participated
        select o.driver_id, c.license, 4007, 3000
        from person p, owns o, car c
        where p.name = 'Jones' and p.driver_id = o.driver_id and
                o.license = c.license and c.model = 'Toyota'
```

c. Delete the Mazda belonging to "John Smith".
Since *model* is not a key of the *car* relation, we can either assume that only one of John Smith's cars is a Mazda, or delete all of John Smith's Mazdas (the query is the same). Again assume *name* is a key for *person*.

```
delete car
where model = 'Mazda' and license in
    (select license
     from person p, owns o
     where p.name = 'John Smith' and p.driver_id = o.driver_id)
```

Note: The *owns*, *accident* and *participated* records associated with the Mazda still exist.

3.6 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.

b. Find the number of students with each grade.

Answer:

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 as
(
select ID,
       case
            when score < 40 then 'F'
            when score < 60 then 'C'
            when score < 80 then 'B'
            else 'A'
       end as grade
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.

3.7 The SQL **like** operator is case sensitive, 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.

Answer:

```
select dept_name
from department
where lower(dept_name) like '%sci%'
```

3.8 Consider the SQL query

branch(*branch_name*, *branch_city*, *assets*)
customer (*customer_name*, *customer_street*, *customer_city*)
loan (*loan_number*, *branch_name*, *amount*)
borrower (*customer_name*, *loan_number*)
account (*account_number*, *branch_name*, *balance*)
depositor (*customer_name*, *account_number*)

Figure 3.1 Banking database for Exercises 3.8 and 3.15.

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 one of $r1$ or $r2$ may be empty.

Answer: 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. Of course if *p* itself is empty, the result is as expected, i.e. empty.

3.9 Consider the bank database of Figure 3.19, where the primary keys are underlined. Construct the following SQL queries for this relational database.

a. Find all customers of the bank who have an account but not a loan.

b. Find the names of all customers who live on the same street and in the same city as "Smith".

c. Find the names of all branches with customers who have an account in the bank and who live in "Harrison".

Answer:

a. Find all customers of the bank who have an account but not a loan.

(**select** *customer_name*
from *depositor*)
except
(**select** *customer_name*
from *borrower*)

The above selects could optionally have **distinct** specified, without changing the result of the query.

b. Find the names of all customers who live on the same street and in the same city as "Smith".
One way of writing the query is as follows.

```
select  F.customer_name
from    customer F join customer S using(customer_street, customer_city)
where   S.customer_name = 'Smith'
```

The join condition could alternatively be specified in the **where** clause, instead of using bf join .. using.

c. Find the names of all branches with customers who have an account in the bank and who live in "Harrison".

```
select  distinct branch_name
from    account natural join depositor natural join customer
where   customer_city = 'Harrison'
```

As usual, the natural join operation could be replaced by specifying join conditions in the **where** clause.

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

a. Find the names and cities of residence of all employees who work for First Bank Corporation.

b. Find the names, street addresses, and cities of residence of all employees who work for First Bank Corporation and earn more than $10,000.

c. Find all employees in the database who do not work for First Bank Corporation.

d. Find all employees in the database who earn more than each employee of Small Bank Corporation.

e. Assume that the companies may be located in several cities. Find all companies located in every city in which Small Bank Corporation is located.

f. Find the company that has the most employees.

g. Find those companies whose employees earn a higher salary, on average, than the average salary at First Bank Corporation.

Answer:

employee (*employee_name*, *street*, *city*)
works (*employee_name*, *company_name*, *salary*)
company (*company_name*, *city*)
manages (*employee_name*, *manager_name*)

Figure 3.20. Employee database.

a. Find the names and cities of residence of all employees who work for First Bank Corporation.

```
select e.employee_name, city
from employee e, works w
where w.company_name = 'First Bank Corporation' and
      w.employee_name = e.employee_name
```

b. Find the names, street address, and cities of residence of all employees who work for First Bank Corporation and earn more than $10,000.
If people may work for several companies, the following solution will only list those who earn more than $10,000 per annum from "First Bank Corporation" alone.

```
select *
from employee
where employee_name in
   (select employee_name
    from works
    where company_name = 'First Bank Corporation' and salary > 10000)
```

As in the solution to the previous query, we can use a join to solve this one also.

c. Find all employees in the database who do not work for First Bank Corporation.
The following solution assumes that all people work for exactly one company.

```
select employee_name
from works
where company_name ≠ 'First Bank Corporation'
```

If one allows people to appear in the database (e.g. in *employee*) but not appear in *works*, or if people may have jobs with more than one company, the solution is slightly more complicated.

```
select employee_name
from employee
where employee_name not in
    (select employee_name
     from works
     where company_name = 'First Bank Corporation')
```

d. Find all employees in the database who earn more than each employee of Small Bank Corporation.

The following solution assumes that all people work for at most one company.

```
select employee_name
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 problem is more complex. It can be solved by using a nested subquery, but we illustrate below how to solve it using the **with** clause.

```
with emp_total_salary as
   (select employee_name, sum(salary) as total_salary
    from works
    group by employee_name
   )
select employee_name
from emp_total_salary
where total_salary > all
   (select total_salary
    from emp_total_salary, works
    where works.company_name = 'Small Bank Corporation' and
          emp_total_salary.employee_name = works.employee_name
   )
```

e. Assume that the companies may be located in several cities. Find all companies located in every city in which Small Bank Corporation is located.

The simplest solution uses the **contains** comparison which was included in the original System R Sequel language but is not present in the subsequent SQL versions.

```
select T.company_name
from company T
where (select R.city
       from company R
       where R.company_name = T.company_name)
    contains
      (select S.city
       from company S
       where S.company_name = 'Small Bank Corporation')
```

Below is a solution using standard SQL.

```
select S.company_name
from company S
where not exists ((select city
                   from company
                   where company_name = 'Small Bank Corporation')
                  except
                   (select city
                    from company T
                    where S.company_name = T.company_name))
```

f. Find the company that has the most employees.

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

g. Find those companies 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')
```

3.11 Consider the relational database of Figure **??**. Give an expression in SQL for each of the following queries.

a. Modify the database so that Jones now lives in Newtown.

b. Give all managers of First Bank Corporation a 10 percent raise unless the salary becomes greater than $100,000; in such cases, give only a 3 percent raise.

Answer:

a. Modify the database so that Jones now lives in Newtown.

The solution assumes that each person has only one tuple in the *employee* relation.

```
update employee
set city = 'Newton'
where person_name = 'Jones'
```

b. Give all managers of First Bank Corporation a 10-percent raise unless the salary becomes greater than $100,000; in such cases, give only a 3-percent raise.

```
update works T
set T.salary = T.salary * 1.03
where T.employee_name in (select manager_name
                          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.employee_name in (select manager_name
                          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
    )
where T.employee_name in (select manager_name
                          from manages) and
      T.company_name = 'First Bank Corporation'
```