CHAPTER 3

Introduction to SQL

Chapter 3 introduces the relational language SQL. Further details of the SQL language are provided in Chapters 4 and 5.

Although our discussion is based on SQL standards, no database system implements the standards exactly as specified, and there are a number of minor syntactic differences that need to be kept in mind. Although we point out some of these differences where required, the system manuals of the database system you use should be used as supplements.

Although it is possible to cover this chapter using only handwritten exercises, we strongly recommend providing access to an actual database system that supports SQL. A style of exercise we have used is to create a moderately large database and give students a list of queries in English to write and run using SQL. We publish the actual answers (that is the result relations they should get, not the SQL they must enter). By using a moderately large database, the probability that a "wrong" SQL query will just happen to return the "right" result relation can be made very small. This approach allows students to check their own answers for correctness immediately rather than wait for grading and thereby it speeds up the learning process. A few such example databases are available on the Web home page of this book, http://db-book.com.

Exercises that pertain to database design are best deferred until after Chapter 8.

Exercises

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

a. Find the names of all students who have taken at least one Comp. Sci. course; make sure there are no duplicate names in the result.

b. Find the IDs and names of all students who have not taken any course offering before Spring 2009.

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.

Answer:

a. SQL query:

```
select  name
from    student natural join takes natural join course
where   course.dept = 'Comp. Sci.'
```

b. SQL query:

```
select  id, name
from    student
except
select  id, name
from    student natural join takes
where   year < 2009
```

Since the **except** operator eliminates duplicates, there is no need to use a **select distinct** clause, although doing so would not affect correctness of the query.

c. SQL query:

```
select  dept, max(salary)
from    instructor
group by dept
```

d. SQL query:

```
select  min(maxsalary)
from    (select dept, max(salary) as maxsalary
         from instructor
         group by dept)
```

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

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

b. Create a section of this course in Autumn 2009, with *section_id* of 1.

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

d. Delete enrollments in the above section where the student's name is Chavez.

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 "database" as a part of the title; ignore case when matching the word with the title.

Answer:

a. SQL query:

```
insert into course
    values ('CS-001', 'Weekly Seminar', 'Comp. Sci.', 0)
```

b. SQL query:

```
insert into section
    values ('CS-001', 1, 'Autumn', 2009, null, null, null)
```

Note that the building, roomnumber and slot were not specified in the question, and we have set them to null. The same effect would be obtained if they were specified to default to null, and we simply omitted values for these attributes in the above insert statement. (Many database systems implicitly set the default value to null, even if not explicitly specified.)

c. SQL query:

```
insert into takes
    select id, 'CS-001', 1, 'Autumn', 2009, null
    from student
    where dept_name = 'Comp. Sci.'
```

d. SQL query:

```
delete from takes
where course_id= 'CS-001' and section_id = 1 and
      year = 2009 and semester = 'Autumn' and
      id in (select id
             from student
             where name = 'Chavez')
```

Note that if there is more than one student named Chavez, all such students would have their enrollments deleted. If we had used = instead of **in**, an error would have resulted if there were more than one student named Chavez.

e. SQL query:

```
delete from takes
where course_id = 'CS-001'

delete from section
where course_id = 'CS-001'

delete from course
where course_id = 'CS-001'
```

If we try to delete the course directly, there will be a foreign key violation because *section* has a foriegn key reference to *course*; similarly, we have to delete corresponding tuples from *takes* before deleting sections, since there is a foreign key reference from *takes* to *section*. As a result of the foreign key violation, the transaction that performs the delete would be rolled back.

f. SQL query:

```
delete from takes
where course_id in
      (select course_id
      from course
      where lower(title) like '%database%')
```

3.13 Write SQL DDL corresponding to the schema in Figure 3.18. Make any reasonable assumptions about data types, and be sure to declare primary and foreign keys.

Answer:

a. SQL query:

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

Figure 3.18 Insurance database for Exercises 3.4 and 3.14.

```
create table person
        (driver_id varchar(50),
        name      varchar(50),
        address   varchar(50),
        primary key (driver_id))
```

b. SQL query:

```
create table car
        (license varchar(50),
        model    varchar(50),
        year     integer,
        primary key (license))
```

c. SQL query:

```
create table accident
        (report_number integer,
        date           date,
        location       varchar(50),
        primary key (report_number))
```

d. SQL query:

```
create table owns
        (driver_id varchar(50),
        license    varchar(50),
        primary key (driver_id,license)
        foriegn key (driver_id) references person
        foriegn key (license) references car)
```

e. SQL query:

```
create table participated
        (report_number integer,
        license         varchar(50),
        driver_id       varchar(50),
        damage_amountinteger,
        primary key (report_number,license)
        foriegn key (license) references car
        foriegn key (report_number) references accident))
```

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

a. Find the number of accidents in which the cars belonging to "John Smith" were involved.

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

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

a. SQL query:

```
select    count (*)
from      accident
where     exists
          (select *
           from participated, owns, person
           where owns.driver_id = person.driver_id
                 and person.name = 'John Smith'
                 and owns.license = participated.license
                 and accident.report_number = participated.report_number)
```

The query can be written in other ways too; for example without a subquery, by using a join and selecting **count**(**distinct** *report_number*) to get a count of number of accidents involving the car.

b. SQL query:

```
update participated
set damage_amount = 3000
where report_number = "AR2197" and
      license = "AABB2000")
```

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

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 3.19 Banking database for Exercises 3.8 and 3.15.

a. Find all customers who have an account at *all* the branches located in "Brooklyn".

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

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

Answer:

a. SQL query:

```
with branchcount as
        (select count(*)
        branch
        where branch_city = 'Brooklyn')
select customer_name
from customer c
where branchcount =
        (select count(distinct branch_name)
        from (customer natural join depositor natural join account
                natural join branch) as d
        where d.customer_name = c.customer_name)
```

There are other ways of writing this query, for example by first finding customers who do not have an account at some branch in Brooklyn, and then removing these customers from the set of all customers by using an **except** clause.

b. SQL query:

```
select sum(amount)
from loan
```

c. SQL query:

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 for Exercises 3.9, 3.10, 3.16, 3.17, and 3.20.

```
select branch_name
from branch
where assets > some
      (select assets
      from branch
      where branch_city = 'Brooklyn')
```

The keyword **any** could be used in place of **some** above.

3.16 Consider the employee database of Figure 3.20, where the primary keys are underlined. Give an expression in SQL for each of the following queries.

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

b. Find all employees in the database who live in the same cities as the companies for which they work.

c. Find all employees in the database who live in the same cities and on the same streets as do their managers.

d. Find all employees who earn more than the average salary of all employees of their company.

e. Find the company that has the smallest payroll.

Answer:

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

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

b. Find all employees in the database who live in the same cities as the companies for which they work.

```
select e.employee_name
from employee e, works w, company c
where e.employee_name = w.employee_name and e.city = c.city and
      w.company_name = c.company _name
```

c. Find all employees in the database who live in the same cities and on the same streets as do their managers.

```
select P.employee_name
from employee P, employee R, manages M
where P.employee_name = M.employee_name and
      M.manager_name = R.employee_name and
      P.street = R.street and P.city = R.city
```

d. Find all employees who earn more than the average salary of all employees of their company.

```
select employee_name
from works T
where salary > (select avg (salary)
                from works S
                where T.company_name = S.company_name)
```

The primary key constraint on *works* ensures that each person works for at most one company.

e. Find the company that has the smallest payroll.

```
select company_name
from works
group by company_name
having sum (salary) <= all (select sum (salary)
                            from works
                            group by company_name)
```

3.17 Consider the relational database of Figure 3.20. Give an expression in SQL for each of the following queries.

a. Give all employees of 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.

Answer:

a. Give all employees of First Bank Corporation a 10-percent raise. (the solution assumes that each person works for at most one company.)

```
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.

```
update works
set salary = salary * 1.1
where employee_name in (select manager_name
                        from manages)
      and company_name = 'First Bank Corporation'
```

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

```
delete from works
where company_name = 'Small Bank Corporation'
```

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

Answer:

- "null" signifies an unknown value.
- "null" is also used when a value does not exist.

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

Answer: Let the set S denote the result of an SQL subquery. We compare (x <> **all** S) with (x **not in** S). If a particular value x_1 satisfies (x_1 <> **all** S) then for all elements y of S $x_1 \neq y$. Thus x_1 is not a member of S and must satisfy (x_1 **not in** S). Similarly, suppose there is a particular value x_2 which satisfies (x_2 **not in** S). It cannot be equal to any element w belonging to S, and hence (x_2 <> **all** S) will be satisfied. Therefore the two expressions are equivalent.

3.20 Give an SQL schema definition for the employee database of Figure 3.20. Choose an appropriate domain for each attribute and an appropriate primary key for each relation schema.

Answer:

```
create table    employee
(employee_name  varchar(20),
 street         char(30),
 city           varchar(20),
 primary key    (employee_name))

create table    works
(employee_name  person_names,
 company_name   varchar(20),
 salary         numeric(8, 2),
 primary key    (employee_name))

create table    company
(company_name   varchar(20),
 city           varchar(20),
 primary key    (company_name))
```

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

Figure 3.21 Library database for Exercise 3.21.

```
create table    manages
(employee_name  varchar(20),
 manager_name   varchar(20),
 primary key    (employee_name))
```

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

a. Print the names of members who have borrowed any book published by "McGraw-Hill".

b. Print the names of members who have borrowed all books published by "McGraw-Hill".

c. For each publisher, print the names of members who have borrowed more than five books of that publisher.

d. Print the average number of books borrowed per member. Take into account that if an member does not borrow any books, then that member does not appear in the *borrowed* relation at all.

Answer:

a. Print the names of members who have borrowed any book published by McGraw-Hill.

```
select name
from member m, book b, borrowed l
where m.memb_no = l.memb_no
      and l.isbn = b.isbn and
            b.publisher = 'McGrawHill'
```

b. Print the names of members who have borrowed all books published by McGraw-Hill. (We assume that all books above refers to all books in the *book* relation.)

```
select distinct m.name
from member m
where not exists
    ((select isbn
    from book
    where publisher = 'McGrawHill')
    except
    (select isbn
    from borrowed l
    where l.memb_no = m.memb_no))
```

c. For each publisher, print the names of members who have borrowed more than five books of that publisher.

```
select publisher, name
from (select publisher, name, count (isbn)
    from member m, book b, borrowed l
    where m.memb_no = l.memb_no
    and l.isbn = b.isbn
    group by publisher, name) as
    membpub(publisher, name, count_books)
where count_books > 5
```

The above query could alternatively be written using the **having** clause.

d. Print the average number of books borrowed per member.

```
with memcount as
    (select count(*)
    from member)
select count(*)/memcount
from borrowed
```

Note that the above query ensures that members who have not borrowed any books are also counted. If we instead used **count(distinct** *memb_no*) from *borrowed*, we would not account for such members.

3.22 Rewrite the **where** clause

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

without using the **unique** construct.

Answer:

```
where(
    (select count(title)
    from course) =
    (select count (distinct title)
    from course))
```

3.23 Consider the query:

```
select course_id, semester, year, section_id, avg (credits_earned)
from takes natural join student
where year = 2009
group by course_id, semester, year, section_id
having count (ID) >= 2;
```

Explain why joining *section* as well in the **from** clause would not change the result.

Answer: The common attributes of *takes* and *section* form a foreign key of *takes*, referencing *section*. As a result, each *takes* tuple would match at most one one *section* tuple, and there would not be any extra tuples in any group. Further, these attributes cannot take on the null value, since they are part of the primary key of *takes*. Thus, joining *section* in the **from** clause would not cause any loss of tuples in any group. As a result, there would be no change in the result.

3.24 Consider the query:

```
with dept_total (dept_name, value) as
    (select dept_name, sum(salary)
    from instructor
    group by dept_name),
dept_total_avg(value) as
    (select avg(value)
    from dept_total)
select dept_name
from dept_total, dept_total_avg
where dept_total.value >= dept_total_avg.value;
```

Rewrite this query without using the **with** construct.

Answer:

There are several ways to write this query. One way is to use subqueries in the where clause, with one of the subqueries having a second level subquery in the from clause as below.

```
select distinct dept_name d
from instructor i
where
    (select sum(salary)
    from instructor
    where department = d)
    >=
    (select avg(s)
    from
        (select sum(salary) as s
        from instructor
        group by department))
```

Note that the original query did not use the *department* relation, and any department with no instructors would not appear in the query result. If we had written the above query using *department* in the outer **from** clause, a department without any instructors could appear in the result if the condition were <= instead of >=, which would not be possible in the original query.

As an alternative, the two subqueries in the where clause could be moved into the from clause, and a join condition (using >=) added.