SQL

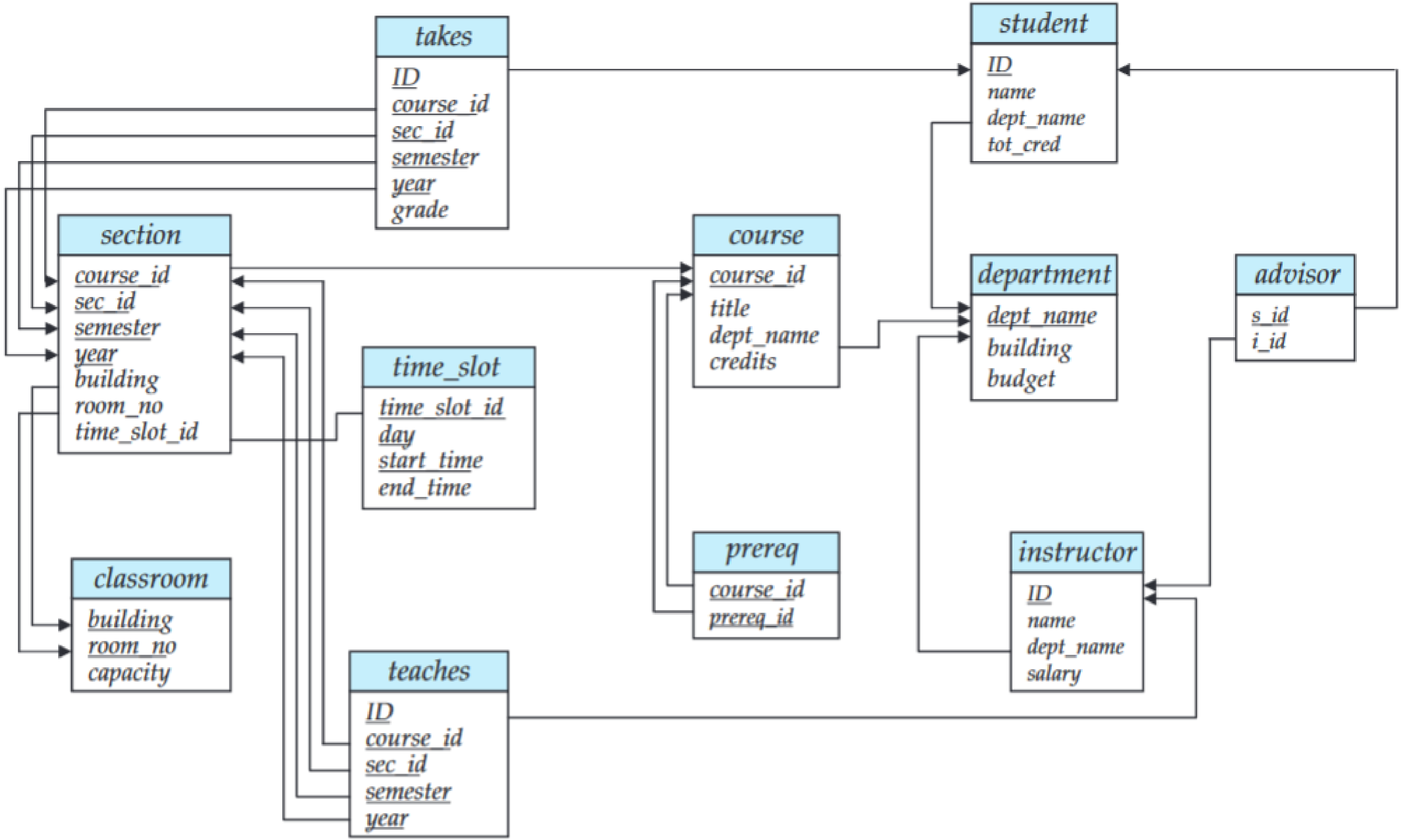

Figure 2.8 Schema diagram for the university database.

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.

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

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.

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.

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).

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.

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.

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,
        (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.

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)
```

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

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.

3.4 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.

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

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).

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.

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

3.6 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 $40 \le score < 60$, grade B if $60 \le score < 80$, and grade A if $80 \le score$. Write SQL queries to do the following:

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

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.

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.

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

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.

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)
```

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%')
```

Employee database

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

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

```
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))
```

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

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.

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'

Library database

member(<u>*memb_no*</u>, *name*, *age*)
book(<u>*isbn*</u>, *title*, *authors*, *publisher*)
borrowed(<u>*memb_no*</u>, <u>*isbn*</u>, *date*)

Consider the library database. 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.

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
```

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*))

3.24 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.