CHAPTER 1

Introduction

Practice Exercises

1.1 This chapter has described several major advantages of a database system. What are two disadvantages?
Answer: Two disadvantages associated with database systems are listed below.

a. Setup of the database system requires more knowledge, money, skills, and time.

b. The complexity of the database may result in poor performance.

1.2 List five ways in which the type declaration system of a language such as Java or C++ differs from the data definition language used in a database.
Answer:

a. Executing an action in the DDL results in the creation of an object in the database; in contrast, a programming language type declaration is simply an abstraction used in the program.

b. Database DDLs allows consistency constraints to be specified, which programming language type systems generally do not allow. These include domain constraints and referential integrity constraints.

c. Database DDLs support authorization, giving different access rights to different users. Programming language type systems do not provide such protection (at best, they protect attributes in a class from being accessed by methods in another class).

d. Programming language type systems are usually much richer than the SQL type system. Most databases support only basic types such as different types of numbers and strings, although some databases do support some complex types such as arrays, and objects.

e. A database DDL is focussed on specifying types of attributes of relations; in contrast, a programming language allows objects, and collections of objects to be created.

1.3 List six major steps that you would take in setting up a database for a particular enterprise.
Answer: Six major steps in setting up a database for a particular enterprise are:

- Define the high level requirements of the enterprise (this step generates a document known as the system requirements specification.)
- Define a model containing all appropriate types of data and data relationships.
- Define the integrity constraints on the data.
- Define the physical level.
- For each known problem to be solved on a regular basis (e.g., tasks to be carried out by clerks or Web users) define a user interface to carry out the task, and write the necessary application programs to implement the user interface.
- Create/initialize the database.

1.4 List at least 3 different types of information that a university would maintain, beyond those listed in Section 1.6.2.
Answer:

- Information about people who are employees of the university but who are not instructors.
- Library information, including books in the library, and who has issued books.
- Accounting information including fee payment, scholarships, salaries, and all other kinds of receipts and payments of the university.

1.5 Suppose you want to build a video site similar to YouTube. Consider each of the points listed in Section 1.2, as disadvantages of keeping data in a file-processing system. Discuss the relevance of each of these points to the storage of actual video data, and to metadata about the video, such as title, the user who uploaded it, tags, and which users viewed it.
Answer:

- **Data redundancy and inconsistency**. This would be relevant to metadata to some extent, although not to the actual video data, which is not updated. There are very few relationships here, and none of them can lead to redundancy.
- **Difficulty in accessing data**. If video data is only accessed through a few predefined interfaces, as is done in video sharing sites today,

this will not be a problem. However, if an organization needs to find video data based on specific search conditions (beyond simple keyword queries) if meta data were stored in files it would be hard to find relevant data without writing application programs. Using a database would be important for the task of finding data.

- **Data isolation**. Since data is not usually updated, but instead newly created, data isolation is not a major issue. Even the task of keeping track of who has viewed what videos is (conceptually) append only, again making isolation not a major issue. However, if authorization is added, there may be some issues of concurrent updates to authorization information.
- **Integrity problems**. It seems unlikely there are significant integrity constraints in this application, except for primary keys. if the data is distributed, there may be issues in enforcing primary key constraints. Integrity problems are probably not a major issue.
- **Atomicity problems**. When a video is uploaded, metadata about the video and the video should be added atomically, otherwise there would be an inconsistency in the data. An underlying recovery mechanism would be required to ensure atomicity in the event of failures.
- **Concurrent-access anomalies**. Since data is not updated, concurrent access anomalies would be unlikely to occur.
- **Security problems**. These would be a issue if the system supported authorization.

1.6 Keyword queries used in Web search are quite different from database queries. List key differences between the two, in terms of the way the queries are specified, and in terms of what is the result of a query.
Answer: Queries used in the Web are specified by providing a list of keywords with no specific syntax. The result is typically an ordered list of URLs, along with snippets of information about the content of the URLs. In contrast, database queries have a specific syntax allowing complex queries to be specified. And in the relational world the result of a query is always a table.

CHAPTER 2

Introduction to the Relational Model

Practice Exercises

2.1 Consider the relational database of Figure **??**. What are the appropriate primary keys?
Answer: The answer is shown in Figure 2.1, with primary keys underlined.

2.2 Consider the foreign key constraint from the *dept_name* attribute of *instructor* to the *department* relation. Give examples of inserts and deletes to these relations, which can cause a violation of the foreign key constraint.
Answer:

- Inserting a tuple:

  (10111, Ostrom, Economics, 110,000)

  into the *instructor* table, where the *department* table does not have the department Economics, would violate the foreign key constraint.

- Deleting the tuple:

  (Biology, Watson, 90000)

  from the *department* table, where at least one student or instructor tuple has *dept_name* as Biology, would violate the foreign key constraint.

$employee\ (\underline{person\_name}, street, city)$
$works\ (\underline{person\_name}, company\_name, salary)$
$company\ (\underline{company\_name}, city)$

Figure 2.1 Relational database for Practice Exercise 2.1.

2.3 Consider the *time_slot* relation. Given that a particular time slot can meet more than once in a week, explain why *day* and *start_time* are part of the primary key of this relation, while *end_time* is not.
Answer: The attributes *day* and *start_time* are part of the primary key since a particular class will most likely meet on several different days, and may even meet more than once in a day. However, *end_time* is not part of the primary key since a particular class that starts at a particular time on a particular day cannot end at more than one time.

2.4 In the instance of *instructor* shown in Figure **??**, no two instructors have the same name. From this, can we conclude that *name* can be used as a superkey (or primary key) of *instructor*?
Answer: No. For this possible instance of the instructor table the names are unique, but in general this may not be always the case (unless the university has a rule that two instructors cannot have the same name, which is a rather unlikey scenario).

2.5 What is the result of first performing the cross product of *student* and *advisor*, and then performing a selection operation on the result with the predicate *s_id* = ID? (Using the symbolic notation of relational algebra, this query can be written as $\sigma_{s\_id=ID}(student \times advisor)$.)
Answer: The result attributes include all attribute values of student followed by all attributes of advisor. The tuples in the result are as follows. For each student who has an advisor, the result has a row containing that students attributes, followed by an *s_id* attribute identical to the students ID attribute, followed by the *i_id* attribute containing the ID of the students advisor.
Students who do not have an advisor will not appear in the result. A student who has more than one advisor will appear a corresponding number of times in the result.

2.6 Consider the following expressions, which use the result of a relational algebra operation as the input to another operation. For each expression, explain in words what the expression does.

a. $\sigma_{year \geq 2009}(takes) \bowtie student$

b. $\sigma_{year \geq 2009}(takes \bowtie student)$

c. $\Pi_{ID, name, course\_id}(student \bowtie takes)$

Answer:

a. For each student who takes at least one course in 2009, display the students information along with the information about what courses the student took. The attributes in the result are:

ID, name, dept_name, tot_cred, course_id, section_id, semester, year, grade

b. Same as (a); selection can be done before the join operation.

c. Provide a list of consisting of

$$ID, name, course\_id$$

of all students who took any course in the university.

2.7 Consider the relational database of Figure **??**. Give an expression in the relational algebra to express each of the following queries:

a. Find the names of all employees who live in city "Miami".

b. Find the names of all employees whose salary is greater than $100,000.

c. Find the names of all employees who live in "Miami" and whose salary is greater than $100,000.

Answer:

a. $\Pi_{name}\ (\sigma_{city\,=\,\text{“Miami”}}\ (employee))$

b. $\Pi_{name}\ (\sigma_{salary\,>\,100000}\ (employee))$

c. $\Pi_{name}\ (\sigma_{city\,=\,\text{“Miami”}\,\wedge\,salary>100000}\ (employee))$

2.8 Consider the bank database of Figure **??**. Give an expression in the relational algebra for each of the following queries.

a. Find the names of all branches located in "Chicago".

b. Find the names of all borrowers who have a loan in branch "Downtown".

Answer:

a. $\Pi_{branch\_name}\ (\sigma_{branch\_city\,=\,\text{“Chicago”}}\ (branch))$

b. $\Pi_{customer\_name}\ (\sigma_{branch\_name\,=\,\text{“Downtown”}}\ (borrower \bowtie loan))$

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*(<u>*grade*</u>, *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 (<u>*driver_id*</u>, *name*, *address*)
car (<u>*license*</u>, *model*, *year*)
accident (<u>*report_number*</u>, <u>*date*</u>, *location*)
owns (<u>*driver_id*</u>, <u>*license*</u>)
participated (<u>*driver_id*</u>, <u>*car*</u>, <u>*report_number*</u>, *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 $40 \leq score < 60$, 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.

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(*<u>branch_name</u>*, *branch_city*, *assets*)
customer (*<u>customer_name</u>*, *customer_street*, *customer_city*)
loan (*<u>loan_number</u>*, *branch_name*, *amount*)
borrower (*<u>customer_name</u>*, *<u>loan_number</u>*)
account (*<u>account_number</u>*, *branch_name*, *balance*)
depositor (*<u>customer_name</u>*, *<u>account_number</u>*)

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 (<u>*employee_name*</u>, *street*, *city*)
works (<u>*employee_name*</u>, *company_name*, *salary*)
company (<u>*company_name*</u>, *city*)
manages (<u>*employee_name*</u>, *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'
```

CHAPTER 4

Intermediate SQL

Practice Exercises

4.1 Write the following queries in SQL:

a. Display a list of all instructors, showing their ID, name, and the number of sections that they have taught. Make sure to show the number of sections as 0 for instructors who have not taught any section. Your query should use an outerjoin, and should not use scalar subqueries.

b. Write the same query as above, but using a scalar subquery, without outerjoin.

c. Display the list of all course sections offered in Spring 2010, along with the names of the instructors teaching the section. If a section has more than one instructor, it should appear as many times in the result as it has instructors. If it does not have any instructor, it should still appear in the result with the instructor name set to "—".

d. Display the list of all departments, with the total number of instructors in each department, without using scalar subqueries. Make sure to correctly handle departments with no instructors.

Answer:

a. Display a list of all instructors, showing their ID, name, and the number of sections that they have taught. Make sure to show the number of sections as 0 for instructors who have not taught any section. Your query should use an outerjoin, and should not use scalar subqueries.

```
select ID, name,
       count(course_id, section_id, year,semester) as 'Number of sections'
from instructor natural left outer join teaches
group by ID, name
```

The above query should not be written using count(*) since count * counts null values also. It could be written using count(*section_id*), or any other attribute from *teaches* which does not occur in *instructor*, which would be correct although it may be confusing to the reader. (Attributes that occur in *instructor* would not be null even if the instructor has not taught any section.)

b. Write the same query as above, but using a scalar subquery, without outerjoin.

```
select ID, name,
    (select count(*) as 'Number of sections'
    from teaches T where T.id = I.id)
from instructor I
```

c. Display the list of all course sections offered in Spring 2010, along with the names of the instructors teaching the section. If a section has more than one instructor, it should appear as many times in the result as it has instructors. If it does not have any instructor, it should still appear in the result with the instructor name set to "—".

```
select course_id, section_id, ID,
    decode(name, NULL, '—', name)
from (section natural left outer join teaches)
    natural left outer join instructor
where semester='Spring' and year= 2010
```

The query may also be written using the **coalesce** operator, by replacing **decode**(..) by **coalesce**(*name*, '—'). A more complex version of the query can be written using union of join result with another query that uses a subquery to find courses that do not match; refer to exercise 4.2.

d. Display the list of all departments, with the total number of instructors in each department, without using scalar subqueries. Make sure to correctly handle departments with no instructors.

```
select dept_name, count(ID)
from department natural left outer join instructor
group by dept_name
```

4.2 Outer join expressions can be computed in SQL without using the SQL **outer join** operation. To illustrate this fact, show how to rewrite each of the following SQL queries without using the **outer join** expression.

a. **select** * **from** *student* **natural left outer join** *takes*

b. **select** * **from** *student* **natural full outer join** *takes*

Answer:

a. **select** * **from** *student* **natural left outer join** *takes*
can be rewritten as:

```
select * from student natural join takes
union
select ID, name, dept_name, tot_cred, NULL, NULL, NULL, NULL, NULL
from student S1 where not exists
    (select ID from takes T1 where T1.id = S1.id)
```

b. **select** * **from** *student* **natural full outer join** *takes*
can be rewritten as:

```
(select * from student natural join takes)
union
(select ID, name, dept_name, tot_cred, NULL, NULL, NULL, NULL, NULL
from student S1
where not exists
    (select ID from takes T1 where T1.id = S1.id))
union
(select ID, NULL, NULL, NULL, course_id, section_id, semester, year, grade
from takes T1
where not exists
    (select ID from student S1 whereT1.id = S1.id))
```

4.3 Suppose we have three relations $r(A, B)$, $s(B, C)$, and $t(B, D)$, with all attributes declared as **not null**. Consider the expressions

- r **natural left outer join** (s **natural left outer join** t), and
- (r **natural left outer join** s) **natural left outer join** t

a. Give instances of relations r, s and t such that in the result of the second expression, attribute C has a null value but attribute D has a non-null value.

b. Is the above pattern, with C null and D not null possible in the result of the first expression? Explain why or why not.

Answer:

a. Consider r = (a,b), s = (b1,c1), t = (b,d). The second expression would give (a,b,NULL,d).

b. It is not possible for D to be not null while C is null in the result of the first expression, since in the subexpression s natural left outer join t, it is not possible for C to be null while D is not null. In the overall expression C can be null if and only if some r tuple does not have a matching B value in s. However in this case D will also be null.

4.4 **Testing SQL queries**: To test if a query specified in English has been correctly written in SQL, the SQL query is typically executed on multiple test databases, and a human checks if the SQL query result on each test database matches the intention of the specification in English.

a. In Section Section 3.3.3The Natural Joinsubsection.3.3.3 we saw an example of an erroneous SQL query which was intended to find which courses had been taught by each instructor; the query computed the natural join of *instructor*, *teaches*, and *course*, and as a result unintentionally equated the *dept_name* attribute of *instructor* and *course*. Give an example of a dataset that would help catch this particular error.

b. When creating test databases, it is important to create tuples in referenced relations that do not have any matching tuple in the referencing relation, for each foreign key. Explain why, using an example query on the university database.

c. When creating test databases, it is important to create tuples with null values for foreign key attributes, provided the attribute is nullable (SQL allows foreign key attributes to take on null values, as long as they are not part of the primary key, and have not been declared as **not null**). Explain why, using an example query on the university database.

Hint: use the queries from Exercise Exercise 4.1Item.138.
Answer:

a. Consider the case where a professor in Physics department teaches an Elec. Eng. course. Even though there is a valid corresponding entry in *teaches*, it is lost in the natural join of *instructor*, *teaches* and *course*, since the instructors department name does not match the department name of the course. A dataset corresponding to the same is:
   instructor = {(12345,'Guass', 'Physics', 10000)}
   teaches = {(12345, 'EE321', 1, 'Spring', 2009)}
   course = {('EE321', 'Magnetism', 'Elec. Eng.', 6)}

b. The query in question 0.a is a good example for this. Instructors who have not taught a single course, should have number of sections as 0 in the query result. (Many other similar examples are possible.)

c. Consider the query

   select * **from** *teaches* **natural join** *instructor*;

   In the above query, we would lose some sections if *teaches*.ID is allowed to be **NULL** and such tuples exist. If, just because *teaches*.ID is a foreign key to *instructor*, we did not create such a tuple, the error in the above query would not be detected.

4.5 Show how to define the view *student_grades* (*ID, GPA*) giving the grade-point average of each student, based on the query in Exercise **??**; recall that we used a relation *grade_points*(<u>*grade*</u>, *points*) to get the numeric points

associated with a letter grade. Make sure your view definition correctly handles the case of *null* values for the *grade* attribute of the *takes* relation.
Answer: We should not add credits for courses with a null grade; further to to correctly handle the case where a student has not completed any course, we should make sure we don't divide by zero, and should instead return a null value.
We break the query into a subquery that finds sum of credits and sum of credit-grade-points, taking null grades into account The outer query divides the above to get the average, taking care of divide by 0.

```
create view student_grades(ID, GPA) as
    select ID, credit_points / decode(credit_sum, 0, NULL, credit_sum)
    from ((select ID, sum(decode(grade, NULL, 0, credits)) as credit_sum,
            sum(decode(grade, NULL, 0, credits*points)) as credit_points
            from(takes natural join course) natural left outer join grade_points
            group by ID)
    union
    select  ID, NULL
    from    student
    where ID not in (select ID from takes))
```

The view defined above takes care of **NULL** grades by considering the creditpoints to be 0, and not adding the corresponding credits in *credit_sum*. The query above ensures that if the student has not taken any course with non-NULL credits, and has *credit_sum* = 0 gets a gpa of **NULL**. This avoid the division by 0, which would otherwise have resulted.
An alternative way of writing the above query would be to use *student* **natural left outer join** *gpa*, in order to consider students who have not taken any course.

4.6 Complete the SQL DDL definition of the university database of Figure Figure 4.8Referential Integrityfigcnt.50 to include the relations *student*, *takes*, *advisor*, and *prereq*.
Answer:

```
create table student
   (ID            varchar (5),
    name          varchar (20) not null,
    dept_name     varchar (20),
    tot_cred      numeric (3,0) check (tot_cred >= 0),
    primary key (ID),
    foreign key (dept_name) references department
                  on delete set null);
```

```
create table takes
   (ID           varchar (5),
    course_id    varchar (8),
    section_id   varchar (8),
    semester     varchar (6),
    year         numeric (4,0),
    grade        varchar (2),
    primary key (ID, course_id, section_id, semester, year),
    foreign key (course_id, section_id, semester, year) references section
                 on delete cascade,
    foreign key (ID) references student
                 on delete cascade);
```

```
create table advisor
   (i_id         varchar (5),
   s_id          varchar (5),
    primary key (s_ID),
    foreign key (i_ID) references instructor (ID)
                 on delete set null,
    foreign key (s_ID) references student (ID)
                 on delete cascade);
```

```
create table prereq
   (course_id    varchar(8),
    prereq_id    varchar(8),
    primary key (course_id, prereq_id),
    foreign key (course_id) references course
                 on delete cascade,
    foreign key (prereq_id) references course);
```

4.7 Consider the relational database of Figure Figure 4.11figcnt.53. Give an SQL DDL definition of this database. Identify referential-integrity constraints that should hold, and include them in the DDL definition.

Answer:

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

```
create table works
    (person_name   char(20),
     company_name char(15),
     salary            integer,
     primary key (person_name),
     foreign key (person_name) references employee,
     foreign key (company_name) references company)


create table company
    (company_name char(15),
     city               char(30),
     primary key (company_name))

ppcreate table manages
    (person_name   char(20),
     manager_name  char(20),
     primary key (person_name),
     foreign key (person_name) references employee,
     foreign key (manager_name) references employee)
```

Note that alternative datatypes are possible. Other choices for **not null** attributes may be acceptable.

4.8 As discussed in Section Section 4.4.7Complex Check Conditions and Assertionssubsection.4.4 we expect the constraint “an instructor cannot teach sections in two different classrooms in a semester in the same time slot” to hold.

a. Write an SQL query that returns all (*instructor*, *section*) combinations that violate this constraint.

b. Write an SQL assertion to enforce this constraint (as discussed in Section Section 4.4.7Complex Check Conditions and Assertionssubsection.4.4.7, current generation database systems do not support such assertions, although they are part of the SQL standard).

Answer:

a.

```
select      ID, name, section_id, semester, year, time_slot_id,
            count(distinct building, room_number)
from        instructor natural join teaches natural join section
group by (ID, name, section_id, semester, year, time_slot_id)
having      count(building, room_number) > 1
```

Note that the **distinct** keyword is required above. This is to allow two different sections to run concurrently in the same time slot and are taught by the same instructor, without being reported as a constraint violation.

b.

```
create assertion check not exists
        ( select ID, name, section_id, semester, year, time_slot_id,
                 count(distinct building, room_number)
        from     instructor natural join teaches natural join section
        group by (ID, name, section_id, semester, year, time_slot_id)
        having   count(building, room_number) > 1)
```

4.9 SQL allows a foreign-key dependency to refer to the same relation, as in the following example:

```
create table manager
    (employee_name   char(20),
     manager_name    char(20),
     primary key employee_name,
     foreign key (manager_name) references manager
                                        on delete cascade )
```

Here, *employee_name* is a key to the table *manager*, meaning that each employee has at most one manager. The foreign-key clause requires that every manager also be an employee. Explain exactly what happens when a tuple in the relation *manager* is deleted.
Answer: The tuples of all employees of the manager, at all levels, get deleted as well! This happens in a series of steps. The initial deletion will trigger deletion of all the tuples corresponding to direct employees of the manager. These deletions will in turn cause deletions of second level employee tuples, and so on, till all direct and indirect employee tuples are deleted.

4.10 SQL-92 provides an n-ary operation called **coalesce**, which is defined as follows: **coalesce**$(A_1, A_2, \ldots, A_n)$ returns the first nonnull A_i in the list $A_1, A_2, \ldots, A_n$, and returns null if all of $A_1, A_2, \ldots, A_n$ are null.
Let a and b be relations with the schemas A(*name, address, title*) and B(*name, address, salary*), respectively. Show how to express a **natural full outer join** b using the **full outer-join** operation with an **on** condition and the **coalesce** operation. Make sure that the result relation does not contain two copies of the attributes *name* and *address*, and that the solution is correct even if some tuples in a and b have null values for attributes *name* or *address*.
Answer:

```
select coalesce(a.name, b.name) as name,
       coalesce(a.address, b.address) as address,
       a.title,
       b.salary
from   a full outer join b on a.name = b.name and
                              a.address = b.address
```

4.11 Some researchers have proposed the concept of *marked* nulls. A marked null $\perp_i$ is equal to itself, but if $i \neq j$, then $\perp_i \neq \perp_j$. One application of marked nulls is to allow certain updates through views. Consider the view *instructor_info* (Section Section 4.2Viewssection.4.2). Show how you can use marked nulls to allow the insertion of the tuple (99999, “Johnson”, “Music”) through *instructor_info*.

Answer: To insert the tuple (99999, “(”Johnson), “Music”) into the view *instructor_info*, we can do the following:

$instructor \leftarrow$ (99999, “Johnson”, $\perp_k$, $\perp$) $\cup$ $instructor$

$department \leftarrow (\perp_k, “Music”, \perp) \cup department$

such that $\perp_k$ is a new marked null not already existing in the database.

Note: “Music” here is the name of a building and may or may not be related to Music department.

CHAPTER 5

Advanced SQL

Practice Exercises

5.1 Describe the circumstances in which you would choose to use embedded SQL rather than SQL alone or only a general-purpose programming language.
Answer: Writing queries in SQL is typically much easier than coding the same queries in a general-purpose programming language. However not all kinds of queries can be written in SQL. Also nondeclarative actions such as printing a report, interacting with a user, or sending the results of a query to a graphical user interface cannot be done from within SQL. Under circumstances in which we want the best of both worlds, we can choose embedded SQL or dynamic SQL, rather than using SQL alone or using only a general-purpose programming language.
Embedded SQL has the advantage of programs being less complicated since it avoids the clutter of the ODBC or JDBC function calls, but requires a specialized preprocessor.

5.2 Write a Java function using JDBC metadata features that takes a `ResultSet` as an input parameter, and prints out the result in tabular form, with appropriate names as column headings.
Answer:

```
public class ResultSetTable implements TabelModel {
    ResultSet result;
    ResultSetMetaData metadata;
    int num_cols;

    ResultSetTable(ResultSet result) throws SQLException {
        this.result = result;
        metadata = result.getMetaData();
        num_cols = metadata.getColumnCount();

        for(int i = 1; i <= num_cols; i++) {
            System.out.print(metadata.getColumnName(i) + ‘‘ ‘‘);
```

```
                }
                System.out.println();
                while(result.next()) {
                    for(int i = 1; i <= num_cols; i++) {
                        System.out.print(result.getString(
                            metadata.getColumnName(i) + ‘‘ ’’));
                    }
                    System.out.println();
                }
            }
        }
```

5.3 Write a Java function using JDBC metadata features that prints a list of all relations in the database, displaying for each relation the names and types of its attributes.
Answer:

```
DatabaseMetaData dbmd = conn.getMetaData();
ResultSet rs = dbmd.getTables();
while (rs.next()) {
    System.out.println(rs.getString(‘‘TABLE_NAME’’);
    ResultSet rs1 = dbmd.getColumns(null, ‘‘schema-name’’,
                rs.getString(‘‘TABLE_NAME’’), ‘‘%’’);
    while (rs1.next()) {
        System.out.println(rs1.getString(‘‘COLUMN_NAME’’),
                rs.getString(‘‘TYPE_NAME’’);
    }
}
```

5.4 Show how to enforce the constraint “an instructor cannot teach in two different classrooms in a semester in the same time slot.” using a trigger (remember that the constraint can be violated by changes to the *teaches* relation as well as to the *section* relation).
Answer: FILL

5.5 Write triggers to enforce the referential integrity constraint from *section* to *time_slot*, on updates to *section*, and *time_slot*. Note that the ones we wrote in Figure 5.8 do not cover the **update** operation.
Answer: FILL

5.6 To maintain the *tot_cred* attribute of the *student* relation, carry out the following:

a. Modify the trigger on updates of *takes*, to handle all updates that can affect the value of *tot_cred*.

b. Write a trigger to handle inserts to the *takes* relation.

c. Under what assumptions is it reasonable not to create triggers on the *course* relation?

Answer: FILL

5.7 Consider the bank database of Figure 5.25. Let us define a view *branch_cust* as follows:

```
create view branch_cust as
    select branch_name, customer_name
    from depositor, account
    where depositor.account_number = account.account_number
```

Suppose that the view is *materialized*; that is, the view is computed and stored. Write triggers to *maintain* the view, that is, to keep it up-to-date on insertions to and deletions from *depositor* or *account*. Do not bother about updates.
Answer: For inserting into the materialized view *branch_cust* we must set a database trigger on an insert into *depositor* and *account*. We assume that the database system uses *immediate* binding for rule execution. Further, assume that the current version of a relation is denoted by the relation name itself, while the set of newly inserted tuples is denoted by qualifying the relation name with the prefix – **inserted**.
The active rules for this insertion are given below –

```
define trigger insert_into_branch_cust_via_depositor
after insert on depositor
referencing new table as inserted for each statement
insert into branch_cust
    select branch_name, customer_name
    from inserted, account
    where inserted.account_number = account.account_number

define trigger insert_into_branch_cust_via_account
after insert on account
referencing new table as inserted for each statement
insert into branch_cust
    select branch_name, customer_name
    from depositor, inserted
    where depositor.account_number = inserted.account_number
```

Note that if the execution binding was *deferred* (instead of immediate), then the result of the join of the set of new tuples of *account* with the set of new tuples of *depositor* would have been inserted by *both* active rules, leading to duplication of the corresponding tuples in *branch_cust*.
The deletion of a tuple from *branch_cust* is similar to insertion, exce pt that a deletion from either *depositor* or *account* will cause the natural join of these relations to have a lesser number of tuples. We denote the newly

deleted set of tuples by qualifying the relation name with the keyword **deleted**.

```
define trigger delete_from_branch_cust_via_depositor
after delete on depositor
referencing old table as deleted for each statement
delete from branch_cust
    select branch_name, customer_name
    from deleted, account
    where deleted.account_number = account.account_number
```

```
define trigger delete_from_branch_cust_via_account
after delete on account
referencing old table as deleted for each statement
delete from branch_cust
    select branch_name, customer_name
    from depositor, deleted
    where depositor.account_number = deleted.account_number
```

5.8 Consider the bank database of Figure 5.25. Write an SQL trigger to carry out the following action: On **delete** of an account, for each owner of the account, check if the owner has any remaining accounts, and if she does not, delete her from the *depositor* relation.
Answer:

```
create trigger check-delete-trigger after delete on  account
referencing old row as orow
for each row
delete from depositor
where depositor.customer_name not in
    ( select customer_name from depositor
    where account_number <> orow.account_number )
end
```

5.9 Show how to express **group by cube**(a, b, c, d) using **rollup**; your answer should have only one **group by** clause.
Answer:

groupby rollup(a), **rollup**(b), **rollup**(c), **rollup**(d)

5.10 Given a relation $S(student, subject, marks)$, write a query to find the top n students by total marks, by using ranking.
Answer: We assume that multiple students do not have the same marks since otherwise the question is not deterministic; the query below deterministically returns all students with the same marks as the n student, so it may return more than n students.

```
select student, sum(marks) as total,
            rank() over (order by (total) desc ) as trank
from S
groupby student
having trank ≤ n
```

5.11 Consider the *sales* relation from Section 5.6. Write an SQL query to compute the cube operation on the relation, giving the relation in Figure 5.21. Do not use the **cube** construct.

Answer:

```
(select color, size, sum(number)
 from sales
 groupby color, size
)
union
(select color, 'all', sum(number)
 from sales
 groupby color
)
union
(select 'all', size, sum(number)
 from sales
 groupby size
)
union
(select 'all', size, sum(number)
 from sales
 groupby size
)
union
(select 'all', 'all', sum(number )
 from sales
)
```

CHAPTER 6 Formal Relational Query Languages

Practice Exercises

6.1 Write the following queries in relational algebra, using the university schema.

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. $\Pi_{title}(\sigma_{dept\_name\ =\ \text{'Comp. Sci'}\ \wedge\ credits=3}(course))$

b. $\Pi_{ID}(\sigma_{IID\ =\ \text{'Einstein'}}(takes \bowtie \rho_{t1(IID,\ course\_id,\ section\_id,\ semester,\ year)}\, teaches))$
Assuming the set version of the relational algebra is used, there is no need to explicitly remove duplicates. If the multiset version is used, the grouping operator can be used without any agggregation to remove duplicates. For example given relation $r(A, B)$ possibly containing duplicates, ${}_{A,B}\mathcal{G}(r)$ would return a duplicate free version of the relation.

c. $\mathcal{G}_{\mathbf{max}(salary)}(instructor)$

d. $instructor \bowtie (\mathcal{G}_{\textbf{max}(salary)\ \textbf{as}\ salary}(instructor))$
Note that the above query renames the maximum salary as salary, so the subsequent natural join outputs only instructors with that salary.

e. ${}_{course\_id,\, section\_id}\mathcal{G}_{\textbf{count}(*)\ \textbf{as}\ enrollment}(\sigma_{year=2009 \wedge semester=\text{Autumn}}(takes))$

f. $t1 \leftarrow {}_{course\_id,\, section\_id}\mathcal{G}_{\textbf{count}(*)\ \textbf{as}\ enrollment}(\sigma_{year=2009 \wedge semester=\text{Autumn}}(takes))$
result = $\mathcal{G}_{\textbf{max}(enrollment)}(t1)$

g. $t2 \leftarrow \mathcal{G}_{\textbf{max}(enrollment)\ \textbf{as}\ enrollment}(t1)$
where $t1$ is as defined in the previous part of the question.
result = $t1 \bowtie t2$

6.2 Consider the relational database of Figure 6.22, where the primary keys are underlined. Give an expression in the relational algebra to express each of the following queries:

a. Find the names of all employees who live in the same city and on the same street as do their managers.

b. Find the names of all employees in this database who do not work for "First Bank Corporation".

c. Find the names of all employees who earn more than every employee of "Small Bank Corporation".

Answer:

a. $\Pi_{person\_name}\ ((employee \bowtie manages)$
$\bowtie_{(manager\_name = employee2.person\_name \wedge employee.street = employee2.street}$
$_{\wedge employee.city = employee2.city)}(\rho_{employee2}\ (employee)))$

b. The following solutions assume that all people work for exactly one company. If one allows people to appear in the database (e.g. in *employee*) but not appear in *works*, the problem is more complicated. We give solutions for this more realistic case later.

$\Pi_{person\_name}\ (\sigma_{company\_name\ \neq\ \text{"First Bank Corporation"}}(works))$

If people may not work for any company:

$\Pi_{person\_name}(employee)\ -\ \Pi_{person\_name}$
$(\sigma_{(company\_name\ =\ \text{"First Bank Corporation"})}(works))$

c. $\Pi_{person\_name}\ (works)\ -\ (\Pi_{works.person\_name}\ (works$
$\bowtie_{(works.salary\ \leq works2.salary\ \wedge\ works2.company\_name = \text{"Small Bank Corporation"})}$
$\rho_{works2}(works)))$

6.3 The natural outer-join operations extend the natural-join operation so that tuples from the participating relations are not lost in the result of the join.

Describe how the theta-join operation can be extended so that tuples from the left, right, or both relations are not lost from the result of a theta join.
Answer:

a. The left outer theta join of $r(R)$ and $s(S)$ ($r \; ⟕_\theta \; s$) can be defined as
$(r \bowtie_\theta s) \cup ((r - \Pi_R(r \bowtie_\theta s)) \times (null, null, \ldots, null))$
The tuple of nulls is of size equal to the number of attributes in S.

b. The right outer theta join of $r(R)$ and $s(S)$ ($r \; ⟖_\theta \; s$) can be defined as
$(r \bowtie_\theta s) \cup ((null, null, \ldots, null) \times (s - \Pi_S(r \bowtie_\theta s)))$
The tuple of nulls is of size equal to the number of attributes in R.

c. The full outer theta join of $r(R)$ and $s(S)$ ($r \; ⟗_\theta \; s$) can be defined as
$(r \bowtie_\theta s) \cup ((null, null, \ldots, null) \times (s - \Pi_S(r \bowtie_\theta s))) \cup$
$((r - \Pi_R(r \bowtie_\theta s)) \times (null, null, \ldots, null))$
The first tuple of nulls is of size equal to the number of attributes in R, and the second one is of size equal to the number of attributes in S.

6.4 (**Division operation**): The division operator of relational algebra, "$\div$", is defined as follows. Let $r(R)$ and $s(S)$ be relations, and let $S \subseteq R$; that is, every attribute of schema S is also in schema R. Then $r \div s$ is a relation on schema $R - S$ (that is, on the schema containing all attributes of schema R that are not in schema S). A tuple t is in $r \div s$ if and only if both of two conditions hold:

- t is in $\Pi_{R-S}(r)$
- For every tuple t_s in s, there is a tuple t_r in r satisfying both of the following:
  a. $t_r[S] = t_s[S]$
  b. $t_r[R - S] = t$

Given the above definition:

a. Write a relational algebra expression using the division operator to find the IDs of all students who have taken all Comp. Sci. courses. (Hint: project *takes* to just ID and *course_id*, and generate the set of all Comp. Sci. *course_id*s using a select expression, before doing the division.)

b. Show how to write the above query in relational algebra, without using division. (By doing so, you would have shown how to define the division operation using the other relational algebra operations.)

Answer:

a. $\Pi_{ID}(\Pi_{ID, course\_id}(takes) \div \Pi_{course\_id}(\sigma_{dept\_name=\text{'Comp. Sci'}}(course))$

b. The required expression is as follows:
$r \leftarrow \Pi_{ID, course\_id}(takes)$

$s \leftarrow \Pi_{course\_id}(\sigma_{dept\_name='\textbf{Comp. Sci}'}(course))$
$\Pi_{ID}(takes) - \Pi_{ID}((\Pi_{ID}(takes) \times s) - r)$

In general, let $r(R)$ and $s(S)$ be given, with $S \subseteq R$. Then we can express the division operation using basic relational algebra operations as follows:

$r \div s = \Pi_{R-S}(r) - \Pi_{R-S}((\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$

To see that this expression is true, we observe that $\Pi_{R-S}(r)$ gives us all tuples t that satisfy the first condition of the definition of division. The expression on the right side of the set difference operator

$$\Pi_{R-S}((\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$$

serves to eliminate those tuples that fail to satisfy the second condition of the definition of division. Let us see how it does so. Consider $\Pi_{R-S}(r) \times s$. This relation is on schema R, and pairs every tuple in $\Pi_{R-S}(r)$ with every tuple in s. The expression $\Pi_{R-S,S}(r)$ merely reorders the attributes of r.

Thus, $(\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r)$ gives us those pairs of tuples from $\Pi_{R-S}(r)$ and s that do not appear in r. If a tuple t_j is in

$$\Pi_{R-S}((\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$$

then there is some tuple t_s in s that does not combine with tuple t_j to form a tuple in r. Thus, t_j holds a value for attributes $R - S$ that does not appear in $r \div s$. It is these values that we eliminate from $\Pi_{R-S}(r)$.

6.5 Let the following relation schemas be given:

$$\begin{aligned} R &= (A, B, C) \\ S &= (D, E, F) \end{aligned}$$

Let relations $r(R)$ and $s(S)$ be given. Give an expression in the tuple relational calculus that is equivalent to each of the following:

a. $\Pi_A(r)$

b. $\sigma_{B=17}(r)$

c. $r \times s$

d. $\Pi_{A,F}(\sigma_{C=D}(r \times s))$

Answer:

a. $\{t \mid \exists q \in r\ (q[A] = t[A])\}$

b. $\{t \mid t \in r \wedge t[B] = 17\}$

c. $\{t \mid \exists p \in r\ \exists q \in s\ (t[A] = p[A] \wedge t[B] = p[B] \wedge t[C] = p[C] \wedge t[D] = q[D]$
$\wedge\, t[E] = q[E] \wedge t[F] = q[F])\}$

d. $\{t \mid \exists\, p \in r\ \exists\, q \in s\ (t[A] = p[A] \land t[F] = q[F] \land p[C] = q[D]\}$

6.6 Let $R = (A, B, C)$, and let r_1 and r_2 both be relations on schema R. Give an expression in the domain relational calculus that is equivalent to each of the following:

a. $\Pi_A(r_1)$

b. $\sigma_{B=17}(r_1)$

c. $r_1 \cup r_2$

d. $r_1 \cap r_2$

e. $r_1 - r_2$

f. $\Pi_{A,B}(r_1) \bowtie \Pi_{B,C}(r_2)$

Answer:

a. $\{< t > \mid \exists\, p, q\ (< t, p, q > \in r_1)\}$

b. $\{< a, b, c > \mid < a, b, c > \in r_1 \land b = 17\}$

c. $\{< a, b, c > \mid < a, b, c > \in r_1 \lor < a, b, c > \in r_2\}$

d. $\{< a, b, c > \mid < a, b, c > \in r_1 \land < a, b, c > \in r_2\}$

e. $\{< a, b, c > \mid < a, b, c > \in r_1 \land < a, b, c > \notin r_2\}$

f. $\{< a, b, c > \mid \exists\, p, q\ (< a, b, p > \in r_1 \land < q, b, c > \in r_2)\}$

6.7 Let $R = (A, B)$ and $S = (A, C)$, and let $r(R)$ and $s(S)$ be relations. Write expressions in relational algebra for each of the following queries:

a. $\{< a > \mid \exists\, b\ (< a, b > \in r \land b = 7)\}$

b. $\{< a, b, c > \mid < a, b > \in r \land < a, c > \in s\}$

c. $\{< a > \mid \exists\, c\ (< a, c > \in s \land \exists\, b_1, b_2\ (< a, b_1 > \in r \land < c, b_2 > \in r \land b_1 > b_2))\}$

Answer:

a. $\Pi_A(\sigma_{B=17}(r))$

b. $r \bowtie s$

c. $\Pi_A(s \bowtie (\Pi_{r.A}(\sigma_{r.b > d.b}(r \times \rho_d(r)))))$

6.8 Consider the relational database of Figure 6.22 where the primary keys are underlined. Give an expression in tuple relational calculus for each of the following queries:

a. Find all employees who work directly for “Jones.”

b. Find all cities of residence of all employees who work directly for “Jones.”

c. Find the name of the manager of the manager of “Jones.”

d. Find those employees who earn more than all employees living in the city “Mumbai.”

Answer:

a.

$$\begin{aligned}\{t \mid \exists\, m \in \mathit{manages}\ (t[\mathit{person\_name}] &= m[\mathit{person\_name}] \\ \wedge\ m[\mathit{manager\_name}] &= \text{'Jones'})\}\end{aligned}$$

b.

$$\begin{aligned}\{t \mid \exists\, m \in \mathit{manages}\ \exists e \in \mathit{employee}(e[\mathit{person\_name}] &= m[\mathit{person\_name}] \\ \wedge\ m[\mathit{manager\_name}] &= \text{'Jones'} \\ \wedge\ t[\mathit{city}] &= e[\mathit{city}])\}\end{aligned}$$

c.

$$\begin{aligned}\{t \mid \exists\, m1 \in \mathit{manages}\ \exists m2 \in \mathit{manages}(m1[\mathit{manager\_name}] &= m2[\mathit{person\_name}] \\ \wedge\ m1[\mathit{person\_name}] &= \text{'Jones'} \\ \wedge\ t[\mathit{manager\_name}] &= m2[\mathit{manager\_name}])\}\end{aligned}$$

d.

$$\begin{aligned}\{t \mid \exists\, w1 \in \mathit{works}\ \neg\exists w2 \in \mathit{works}(w1[\mathit{salary}] &< w2[\mathit{salary}] \\ \exists e2 \in \mathit{employee}\ (w2[\mathit{person\_name}] &= e2[\mathit{person\_name}] \\ \wedge\ e2[\mathit{city}] &= \text{'Mumbai'}))\}\end{aligned}$$

6.9 Describe how to translate join expressions in SQL to relational algebra.
Answer: A query of the form

select $A1, A2, \ldots, An$
from $R1, R2, \ldots, Rm$
where P

can be translated into relational algebra as follows:
$\Pi_{A1, A2, \ldots, An}(\sigma_P(R1 \times R2 \times \ldots \times Rm))$
An SQL join expression of the form

$R1$ **natural join** $R2$

can be written as $R1 \bowtie R2$.
An SQL join expression of the form

$R1$ **join** $R2$ **on** (P)

can be written as $R1 \bowtie_P R2$.

An SQL join expression of the form

$$R1 \textbf{ join } R2 \textbf{ using } (A1, A2, \ldots, An)$$

can be written as $\Pi_S(R1 \bowtie_{R1.A1=R2.A1 \,\wedge\, R1.A2=R2.A2 \,\wedge\, \ldots\, R1.An=R2.An} R2)$
where S is $A1, A2, \ldots, An$ followed by all attributes of $R1$ other than $R1.A1, R1.A2, \ldots, R1.An$, followed by all attributes of $R2$ other than $R2.A1, R2.A2, \ldots, R2.An$,
The outer join versions of the SQL join expressions can be similarly written by using ⟕, ⟖ and ⟗ in place of $\bowtie$.[1]
The most direct way to handle subqueries is to extend the relational algebra. To handle where clause subqueries, we need to allow selection predicates to contain nested relational algebra expressions, which can reference correlation attributes from outer level relations. Scalar subqueries can be similarly translated by allowing nested relational algebra expressions to appear in scalar expressions. An alternative approach to handling such subqueries used in some database systems, such as Microsoft SQL Server, introduces a new relational algebra operator called the Apply operator; see Chapter 30, page 1230-1231 for details. Without such extensions, translating subqueries into standard relational algebra can be rather complicated.

[1]The case of outer joins with the **using** clause is a little more complicated; with a right outer join it is possible that $R1.A1$ is null, but $R2.A1$ is not, and the output should contain the non-null value. The SQL **coalesce** function can be used, replacing S by **coalesce**($R1.A1$, $R2.A1$), **coalesce**($R1.A2$, $R2.A2$), ... **coalesce**($R1.An$, $R2.An$), followed by the other attributes of $R1$ and $R2$.

CHAPTER 7

Database Design and the E-R Model

Practice Exercises

7.1 **Answer:** The E-R diagram is shown in Figure 7.1. Payments are modeled as weak entities since they are related to a specific policy.
Note that the participation of accident in the relationship *participated* is not total, since it is possible that there is an accident report where the participating car is unknown.

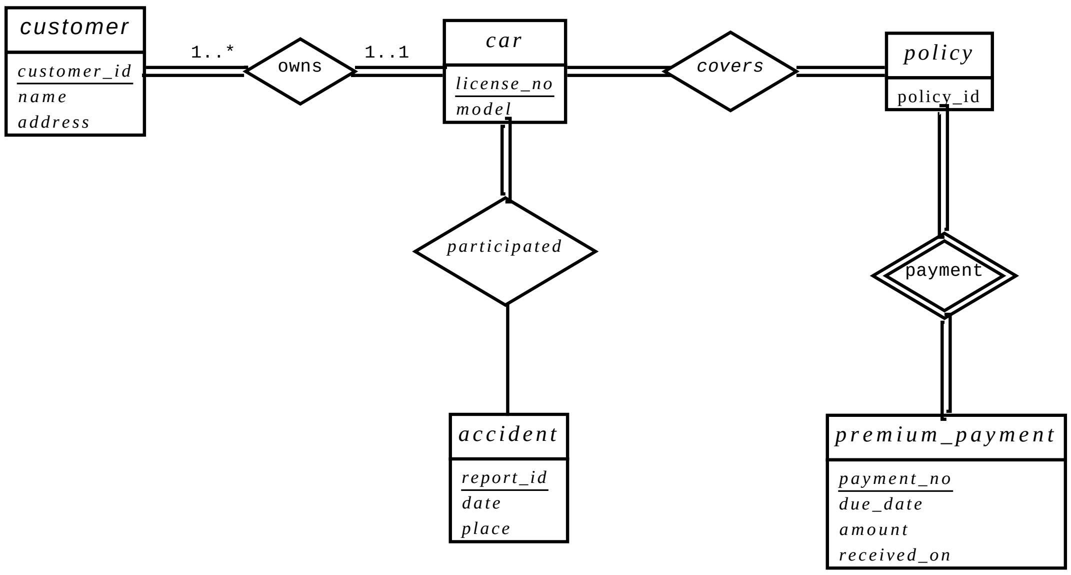

Figure 7.1 E-R diagram for a car insurance company.

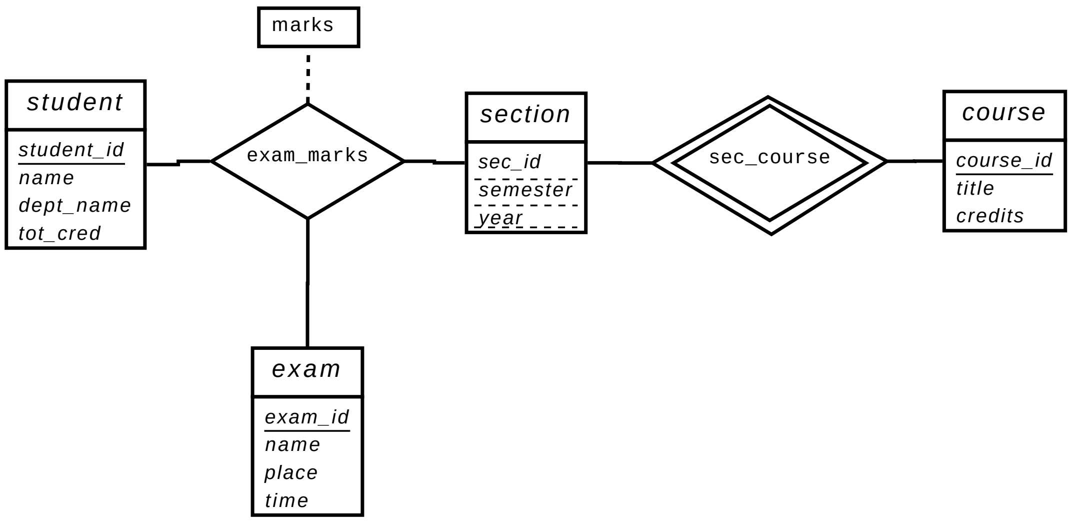

Figure 7.2 E-R diagram for marks database.

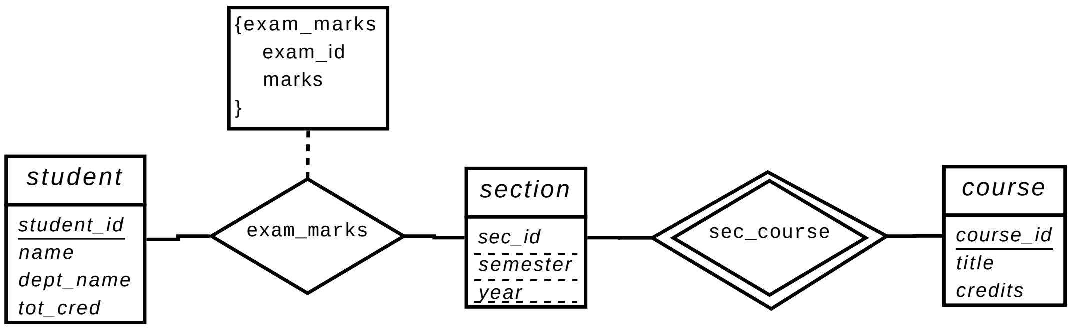

Figure 7.3 Another E-R diagram for marks database.

7.2 **Answer:** Note: the name of the relationship "course offering" needs to be changed to "section".

a. The E-R diagram is shown in Figure 7.2. Note that an alterantive is to model examinations as weak entities related to a section, rather than as a strong entity. The marks relationship would then be a binary relationship between *student* and *exam*, without directly involving *section*.

b. The E-R diagram is shown in Figure 7.3. Note that here we have not modeled the name, place and time of the exam as part of the relationship attributes. Doing so would result in duplication of the information, once per student, and we would not be able to record this information without an associated student. If we wish to represent this information, we would need to retain a separate entity corresponding to each exam.

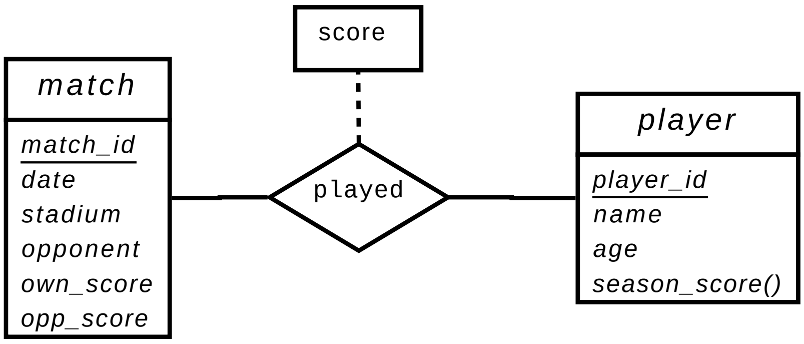

Figure 7.4 E-R diagram for favourite team statistics.

7.3 **Answer:** The diagram is shown in Figure 7.4.

7.4 **Answer:** The different occurrences of an entity may have different sets of attributes, leading to an inconsistent diagram. Instead, the attributes of an entity should be specified only once. All other occurrences of the entity should omit attributes. Since it is not possible to have an entity without any attributes, an occurrence of an entity without attributes clearly indicates that the attributes are specified elsewhere.

7.5 **Answer:**

a. If a pair of entity sets are connected by a path in an E-R diagram, the entity sets are related, though perhaps indirectly. A disconnected graph implies that there are pairs of entity sets that are unrelated to each other. In an enterprise, we can say that the two departments are completely independent of each other. If we split the graph into connected components, we have, in effect, a separate database corresponding to each connected component.

b. As indicated in the answer to the previous part, a path in the graph between a pair of entity sets indicates a (possibly indirect) relationship between the two entity sets. If there is a cycle in the graph then every pair of entity sets on the cycle are related to each other in at least two distinct ways. If the E-R diagram is acyclic then there is a unique path between every pair of entity sets and, thus, a unique relationship between every pair of entity sets.

7.6 **Answer:**

a. Let $E = \{e_1, e_2\}$, $A = \{a_1, a_2\}$, $B = \{b_1\}$, $C = \{c_1\}$, $R_A = \{(e_1, a_1), (e_2, a_2)\}$, $R_B = \{(e_1, b_1)\}$, and $R_C = \{(e_1, c_1)\}$. We see that because of the tuple (e_2, a_2), no instance of R exists which corresponds to E, R_A, R_B and R_C.

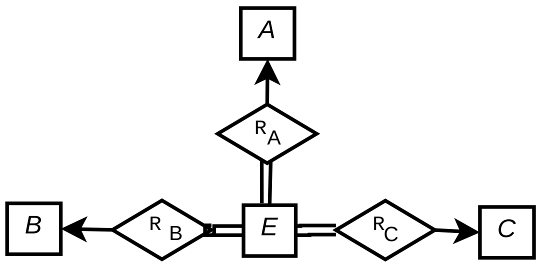

Figure 7.5 E-R diagram for Exercise 7.6b.

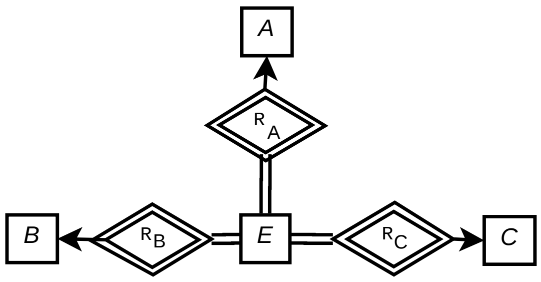

Figure 7.6 E-R diagram for Exercise 7.6d.

b. See Figure 7.5. The idea is to introduce total participation constraints between E and the relationships R_A, R_B, R_C so that every tuple in E has a relationship with A, B and C.

c. Suppose A totally participates in the relationhip R, then introduce a total participation constraint between A and R_A.

d. Consider E as a weak entity set and R_A, R_B and R_C as its identifying relationship sets. See Figure 7.6.

7.7 Answer: The primary key of a weak entity set can be inferred from its relationship with the strong entity set. If we add primary key attributes to the weak entity set, they will be present in both the entity set and the relationship set and they have to be the same. Hence there will be redundancy.

7.8 Answer: In this example, the primary key of *section* consists of the attributes (*course_id*, *semester*, *year*), which would also be the primary key of *sec_course*, while *course_id* is a foreign key from *sec_course* referencing *course*. These constraints ensure that a particular *section* can only correspond to one *course*, and thus the many-to-one cardinality constraint is enforced. However, these constraints cannot enforce a total participation constraint, since a course or a section may not participate in the *sec_course* relationship.

7.9 Answer:

In addition to declaring *s_ID* as primary key for *advisor*, we declare *i_ID* as a super key for *advisor* (this can be done in SQL using the **unique** constraint on *i_ID*).

7.10 **Answer:** The foreign key attribute in R corresponding to primary key of B should be made **not null**. This ensures that no tuple of A which is not related to any entry in B under R can come in R. For example, say **a** is a tuple in A which has no corresponding entry in R. This means when R is combined with A, it would have foreign key attribute corresponding to B as **null** which is not allowed.

7.11 **Answer:**

a. For the many-to-many case, the relationship must be represented as a separate relation which cannot be combined with either participating entity. Now, there is no way in SQL to ensure that a primary key value occurring in an entity $E1$ also occurs in a many-to-many relationship R, since the corresponding attribute in R is not unique; SQL foreign keys can only refer to the primary key or some other unique key.
Similarly, for the one-to-many case, there is no way to ensure that an attribute on the one side appears in the relation corresponding to the many side, for the same reason.

b. Let the relation R be many-to-one from entity A to entity B with a and b as their respective primary keys, repectively. We can put the following check constraints on the "one" side relation B:

constraint *total_part* **check** (*b* in (**select** *b* **from** *A*));
set constraints *total_part* **deferred**;

Note that the constraint should be set to deferred so that it is only checked at the end of the transaction; otherwise if we insert a *b* value in B before it is inserted in A the above constraint would be violated, and if we insert it in A before we insert it in B, a foreign key violation would occur.

7.12 **Answer:** A inherits all the attributes of X plus it may define its own attributes. Similarly C inherits all the attributes of Y plus its own attributes. B inherits the attributes of both X and Y. If there is some attribute *name* which belongs to both X and Y, it may be referred to in B by the qualified name *X.name* or *Y.name*.

7.13 **Answer:**

a. The E-R diagram is shown in Figure 7.7. The primary key attributes *student_id* and *instructor_id* are assumed to be immutable, that is they are not allowed to change with time. All other attributes are assumed to potentially change with time.
Note that the diagram uses multivalued composite attributes such as *valid_times* or *name*, with sub attributes such as *start_time* or *value*.

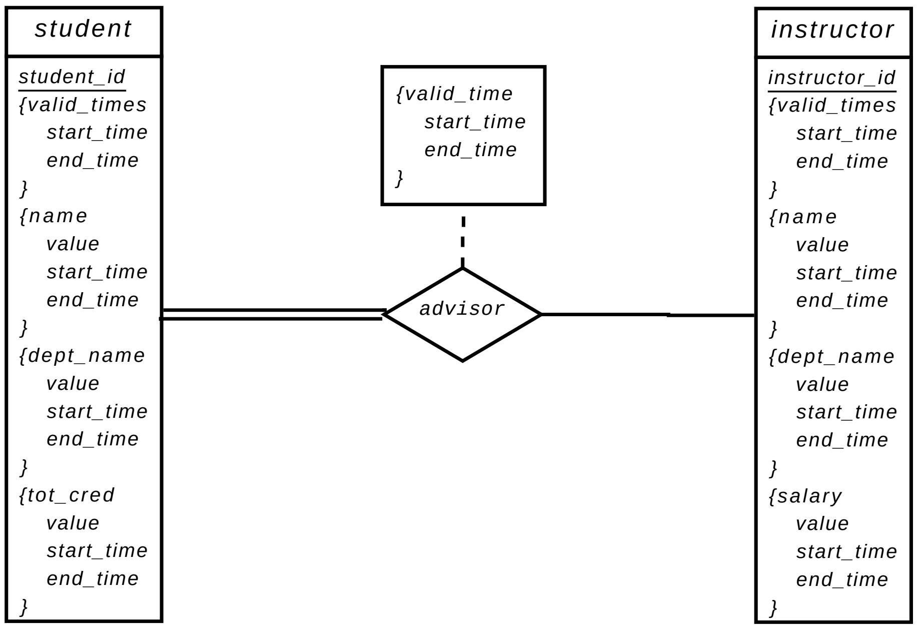

Figure 7.7 E-R diagram for Exercise 7.13

The *value* attribute is a subattribute of several attributes such as *name*, *tot_cred* and *salary*, and refers to the name, total credits or salary during a particular interval of time.

b. The generated relations are as shown below. Each multivalued attribute has turned into a relation, with the relation name consisting of the original relation name concatenated with the name of the multivalued attribute. The relation corresponding to the entity has only the primary key attribute.

student(student_id)
student_valid_times(student_id, start_time, end_time)
student_name(student_id, value, start_time, end_time
student_dept_name(student_id, value, start_time, end_time
student_tot_cred(student_id, value, start_time, end_time
instructor(instructor_id)
instructor_valid_times(instructor_id, start_time, end_time)
instructor_name(instructor_id, value, start_time, end_time
instructor_dept_name(instructor_id, value, start_time, end_time
instructor_salary(instructor_id, value, start_time, end_time
advisor(student_id, intructor_id, start_time, end_time)

The primary keys shown are derived directly from the E-R diagram. If we add the additional constraint that time intervals cannot overlap (or even the weaker condition that one start time cannot have two end times), we can remove the *end_time* from all the above primary keys.