CHAPTER 4

Intermediate SQL

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* **where***T1.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 **??** 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 **??**.

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*(*grade*, *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 **??** 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 **??**. Give an SQL DDL definition of this database. Identify referential-integrity constraints that should hold, and include them in the DDL definition.

Answer: