CHAPTER 4

Intermediate SQL

Practice Exercises

4.1 Consider the following SQL query that seeks to find a list of titles of all courses taught in Spring 2017 along with the name of the instructor.

> **select** name, title
> **from** *instructor* **natural join** *teaches* **natural join** *section* **natural join** *course*
> **where** *semester* = 'Spring' **and** year = 2017

What is wrong with this query?

Answer:

Although the query is syntactically correct, it does not compute the expected answer because *dept_name* is an attribute of both *course* and *instructor*. As a result of the natural join, results are shown only when an instructor teaches a course in her or his own department.

4.2 Write the following queries in SQL:

a. Display a list of all instructors, showing each instructor's ID and the number of sections taught. Make sure to show the number of sections as 0 for instructors who have not taught any section. Your query should use an outer join, and should not use subqueries.

b. Write the same query as in part a, but using a scalar subquery and not using outer join.

c. Display the list of all course sections offered in Spring 2018, along with the ID and name of each instructor teaching the section. If a section has more than one instructor, that section should appear as many times in the result as it has instructors. If a section 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 subqueries. Make sure to show departments that have no instructors, and list those departments with an instructor count of zero.

Answer:

a. Display a list of all instructors, showing each instructor's ID and the number of sections taught. Make sure to show the number of sections as 0 for instructors who have not taught any section. Your query should use an outer join, and should not use subqueries.

> **select** *ID*, **count**(*sec_id*) **as** Number_of_sections
> **from** *instructor* **natural left outer join** *teaches*
> **group by** *ID*

The above query should not be written using count(*) since that would count null values also. It could be written using any 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, and not using outerjoin.

> **select** *ID*,
>   (**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 2018, along with the ID and name of each instructor teaching the section. If a section has more than one instructor, that section should appear as many times in the result as it has instructors. If a section does not have any instructor, it should still appear in the result with the instructor name set to "−".

> **select** *course_id*, *sec_id*, *ID*,
>   **decode**(*name*, *null*, '−', *name*) **as** name
> **from** (*section* **natural left outer join** *teaches*)
>   **natural left outer join** *instructor*
> **where** *semester='Spring'* and *year*= 2018

The query may also be written using the **coalesce** operator, by replacing **decode**(..) with **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 4.3.

d. Display the list of all departments, with the total number of instructors in each department, without using subqueries. Make sure to show departments that have no instructors, and list those departments with an instructor count of zero.

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

4.3 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*, *sec_id*, *semester*, *year*, *grade*
from *takes T1*
where not exists
(**select** *ID* **from** *student S1* **where***T1.id* = *S1.id*))

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

a. Give instances of relations r, s, and t such that in the result of
(r **natural left outer join** s) **natural left outer join** t
attribute C has a null value but attribute D has a non-null value.

b. Are there instances of r, s, and t such that the result of
r **natural left outer join** (s **natural left outer join** t)
has a null value for C but a non-null value for D? 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. Since s **natural left outer join** t is computed first, the absence of nulls is both s and t implies that each tuple of the result can have D null, but C can never be null.

4.5 **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 4.1.1 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 it 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 4.2.

Answer:

a. Consider the case where a professor in the 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 instructor's department name does not match the department name of the course. A dataset corresponding to the same is:

instructor = {('12345','Gauss', 'Physics', 10000)}
teaches = {('12345', 'EE321', 1, 'Spring', 2017)}
course = {('EE321', 'Magnetism', 'Elec. Eng.', 6)}

b. The query in question 4.2(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 this 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.

4.6 Show how to define the view *student_grades* (*ID, GPA*) giving the grade-point average of each student, based on the query in Exercise 3.2; 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 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 zero.

```
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, null
    from    student
    where   ID not in (select ID from takes))
```

The view defined above takes care of *null* grades by considering the credit points to be 0 and not adding the corresponding credits in *credit_sum*.

employee (*ID*, *person_name*, *street*, *city*)
works (*ID*, *company_name*, *salary*)
company (*company_name*, *city*)
manages (*ID*, *manager_id*)

Figure 4.12 Employee database.

The query above ensures that a student who has not taken any course with non-null credits, and has *credit_sum* = 0 gets a GPA of *null*. This avoids the division by zero, which would otherwise have resulted.

In systems that do note support **decode**, an alternative is the **case** construct. Using **case**, the solution would be written as follows:

```
create view student_grades(ID, GPA) as
    select ID, credit_points / (case when credit_sum = 0 then null
                            else credit_sum end)
    from ((select ID, sum (case when grade is null then 0
                            else credits end) as credit_sum,
                            sum (case when grade is null then 0
                            else credits*points end) as credit_points
          from(takes natural join course) natural left outer join grade_points
          group by ID)
    union
    select  ID, null, null
    from    student
    where   ID not in (select ID from takes))
```

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.

4.7 Consider the employee database of Figure 4.12. Give an SQL DDL definition of this database. Identify referential-integrity constraints that should hold, and include them in the DDL definition.

Answer:
Plese see **??**.

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

4.8 As discussed in Section 4.4.8, we expect the constraint "an instructor cannot teach sections in two different classrooms in a semester in the same time slot" to hold.

```
create table employee
    (ID             numeric(6,0),
     person_name    char(20),
     street         char(30),
     city           char(30),
     primary key (ID))
```

```
create table works
    (ID             numeric(6,0),
     company_name   char(15),
     salary         integer,
     primary key (ID),
     foreign key (ID) references employee,
     foreign key (company_name) references company)
```

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

```
create table manages
    (ID             numeric(6,0),
     manager_iid    numeric(6,0),
     primary key (ID),
     foreign key (ID) references employee,
     foreign key (manager_iid) references employee(ID))
```

Figure 4.101 Figure for Exercise 4.7.

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 4.4.8, current generation database systems do not support such assertions, although they are part of the SQL standard).

Answer:

a. Query:

```
select   ID, name, sec_id, semester, year, time_slot_id,
         count(distinct building, room_number)
from     instructor natural join teaches natural join section
group by (ID, name, sec_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. Query:

```
create assertion check not exists
    ( select ID, name, sec_id, semester, year, time_slot_id,
             count(distinct building, room_number)
    from     instructor natural join teaches natural join section
    group by (ID, name, sec_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_ID      char(20),
     manager_ID       char(20),
     primary key employee_ID,
     foreign key (manager_ID) references manager(employee_ID)
                                  on delete cascade )
```

Here, *employee_ID* 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 Given the relations *a*(*name, address, title*) and *b*(*name, address, salary*), show how to express *a* **natural full outer join** *b* using the **full outer-join** operation with an **on** condition rather than using the **natural join** syntax. This can be done using 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 Operating systems usually offer only two types of authorization control for data files: read access and write access. Why do database systems offer so many kinds of authorization?

Answer: There are many reasons—we list a few here. One might wish to allow a user only to append new information without altering old information. One might wish to allow a user to access a relation but not change its schema. One might wish to limit access to aspects of the database that are not technically data access but instead impact resource utilization, such as creating an index.

4.12 Suppose a user wants to grant **select** access on a relation to another user. Why should the user include (or not include) the clause **granted by current role** in the **grant** statement?

Answer: Both cases give the same authorization at the time the statement is executed, but the long-term effects differ. If the grant is done based on the role, then the grant remains in effect even if the user who performed the grant leaves and that user's account is terminated. Whether that is a good or bad idea depends on the specific situation, but usually granting through a role is more consistent with a well-run enterprise.

4.13 Consider a view *v* whose definition references only relation *r*.

- If a user is granted **select** authorization on *v*, does that user need to have **select** authorization on *r* as well? Why or why not?
- If a user is granted **update** authorization on *v*, does that user need to have **update** authorization on *r* as well? Why or why not?
- Give an example of an **insert** operation on a view *v* to add a tuple *t* that is not visible in the result of **select** * **from** *v*. Explain your answer.

Answer:

- No. This allows a user to be granted access to only part of relation *r*.

- Yes. A valid update issued using view *v* must update *r* for the update to be stored in the database.
- Any tuple *t* compatible with the schema for *v* but not satisfying the **where** clause in the definition of view *v* is a valid example. One such example appears in Section 4.2.4.